UNIT – V
POINTERS

TOPICS

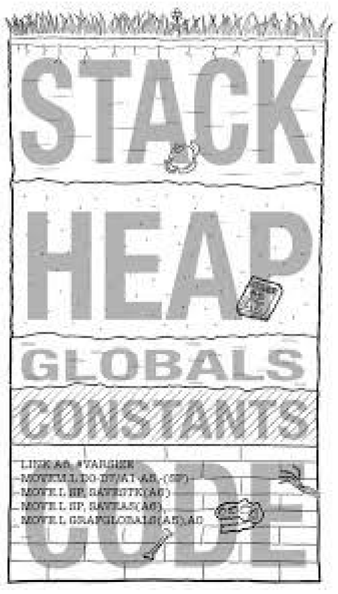
Introduction of pointers
Declaration and initializing of pointers
Address and indirection operator
Pointer expressions and Pointer arithmetic
Pointers to Pointers
void pointer
NULL pointer
dangling pointer
const qualifier
Arrays and pointers
- i. array and pointers
- ii. arr+1 vs &arr+1
- iii. Accessing array elements using pointers
- iv. pointer to array
- v. array of pointer
- vi. 2D array and pointer

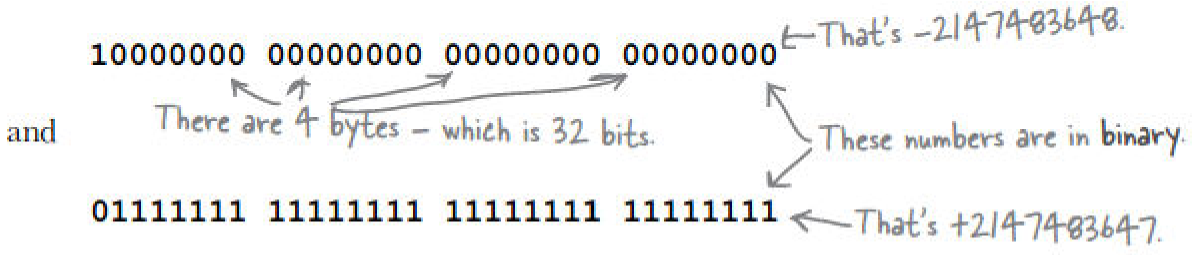
String and pointers
- i. string and pointer
- ii. Accessing string using pointer
- iii. %s and string

Pointers and functions
- i. Pointer to function
- ii. Passing pointer to function
- iii. call by value
- iv. call by reference

Dynamic memory allocation
- i. malloc function
- ii. calloc function
- iii. malloc vs calloc
- iv. realloc
- v. free

Command line arguments
Programming examples.

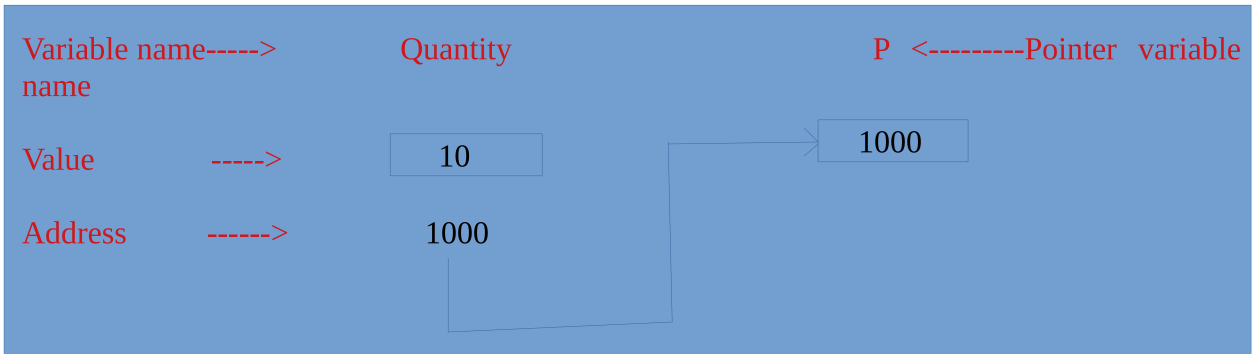




Range of int

POINTERS

- Pointer data type is one of the strengths of the C language.
- The pointer is a powerful technique to access the data by indirect reference as it holds the address of the variable where it has been stored in the memory.
- **A pointer is a variable which holds the memory address of another variable.**
- Sometimes, only with the pointer a complex data type can be declared and accessed easily.



The pointers has the following advantages:

1. Pointers are more efficient in handling **arrays** and data tables.
2. Pointers can be used to return multiple values from a function via **function arguments**.
3. Pointer permit **references to functions** and thereby facilitating passing of functions as **arguments to other functions.**
4. The use of **pointer arrays to character strings** results in saving of data storage space in memory.
5. Pointers allow C to support dynamic memory management.
6. Pointers provide an efficient tool for manipulating dynamic data structures such as **structures**, **linked lists**, **queues**, stacks, and **trees**.
7. Pointers **reduce length** and complexity of programs.
8. They increase the execution speed and thus **reduce the program execution time.**

DECLARATION AND INITIALIZATION OF POINTER VARIABLES

POINTER DEFINITION:

- C provides data manipulation with addresses of variables therefore execution time is reduced.
- Such concept is possible with special data type called pointer.
- **A pointer is a variable which holds the address of another variable or identifier this allows indirect access of data**

1. DECLARING POINTER VARIABLES:

- In C, every variable must be declared for its type.
- Since pointer variables contain addresses that belong to a seperate data type, they must be declared as pointers before we use them.
- The declaration of a pointer variable takes the following form:

```
data_type  *pt_name;
```

This tells the compiler three things about the variable **pt_name.**

1. The asterick(*) tells that the variable **pt_name** is a pointer variable.
2. **pt_name** needs a memory location
3. **pt_name** points to a variable of type data_type.

For example,

```
int *p;   //integer pointer
```

declares the variable p as a pointer variable that points to an integer data type.

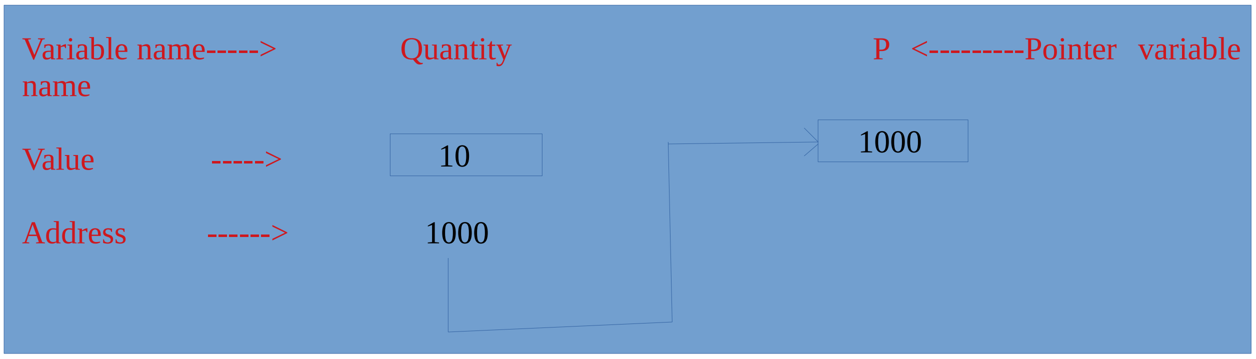
2. INITIALIZATION OF POINTER VARIABLES:

- The process of assigning the address of a variable to a pointer variable is known as initialization.
- The programs with uninitialized pointers will produce erroneous results.
- It is therefore important to initialize pointer variables carefully before they are used in the program.
- Once a pointer variable has been declared, we can use the assignment operator to initialize the variable.

Example:

```
int quantity=10;
int *p; //declaration
p=&quantity;//initialization
```

The only requirement here is that the variable **quantity** must be declared before the initialization takes place.


Variable name-----> name
Quantity
P <---------Pointer variable
Value ----->
10
1000
Address ------>
1000

ACCESSING THE ADDRESS OF A VARIABLE:

- The actual location of a variable in the memory is system dependent and therefore, the address of a variable is not known to us immediately.
- This can be done with the help of the **operator &** available in C.
- The **operator &** immediately preceding a variable returns the address of the variable associated with it.

For example, the statement

```
p=&quantity;
```

would assign the address 1000(the location of **quantity**) to the varaiable p. The & operator can be remembered as **‘address of’**.

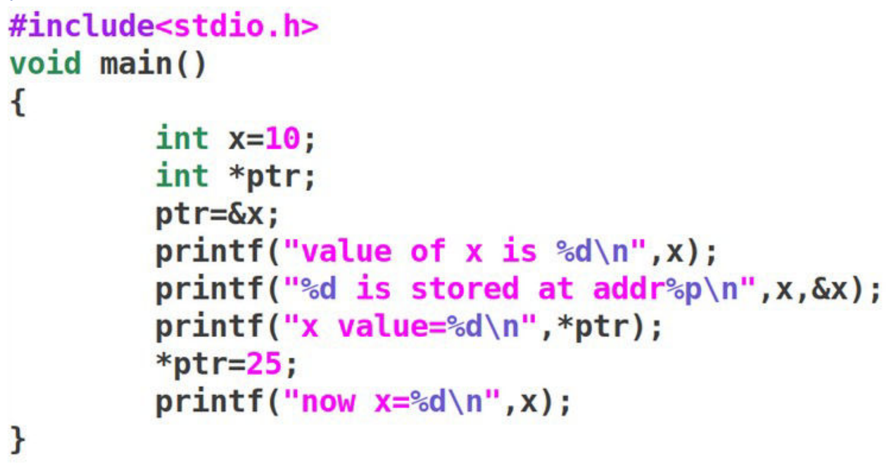
ACCESSING A VARIABLE THROUGH ITS POINTER(*)

- Once a pointer has been assigned the address of a variable, to access the value of the variable using another unary operator *(asterisk), usually known as the *indirection operator.*
- Another name for the indirection operator is the *dereferencing operator.*
- Consider the following statements.

```
int quantity=179;
int *p,n;
p=&quantity;
n=*p;
```

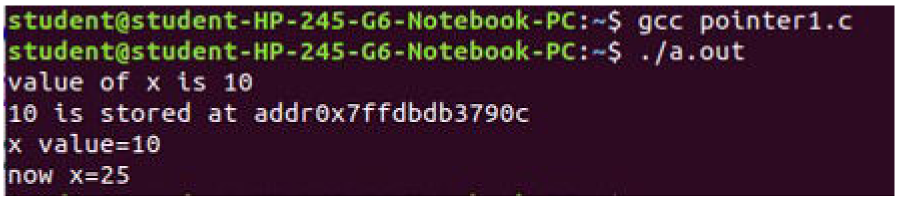
- In this case, *p returns the value of the variable **quantity**, because **p** is the address of **quantity.** The * can be remembered as ‘value at address’.
- Thus the value of n would be 179.

PROGRAM:



```
#include<stdio.h>
void main()
{
        int x=10;
        int *ptr;
        ptr=&x;
        printf("value of x is %d\n",x);
        printf("%d is stored at addr%p\n",x,&x);
        printf("x value=%d\n",*ptr);
        *ptr=25;
        printf("now x=%d\n",x);
}
```

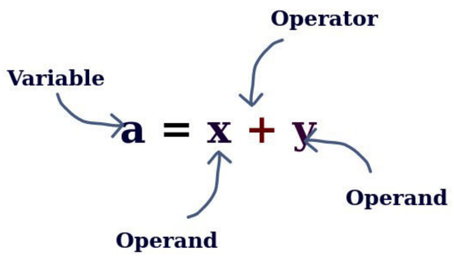
OUTPUT:



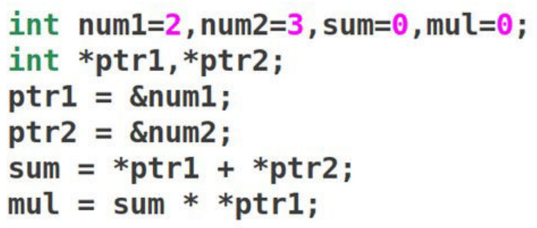
```
student@student-HP-245-G6-Notebook-PC:~$ gcc pointer1.c
student@student-HP-245-G6-Notebook-PC:~$ ./a.out
value of x is 10
10 is stored at addr0x7ffdbdb3790c
x value=10
now x=25
```

POINTER EXPRESSIONS:

- An `expression` in C is a combination of *operands* and *operators* – it computes a single value stored in a *variable*.
- The operator denotes the action or operation to be performed.
- The operands are the items to which we apply the operation.



- Similar to other variables, pointer varaiables can also be used in expressions.
- For example, if ptr1 and ptr2 are pointers, then ptr1 and ptr2 are pointers, then the following statements are valid.



```
int num1=2,num2=3,sum=0,mul=0;
int *ptr1,*ptr2;
ptr1 = &num1;
ptr2 = &num2;
sum = *ptr1 + *ptr2;
mul = sum * *ptr1;
```

POINTER ARITHMETIC:

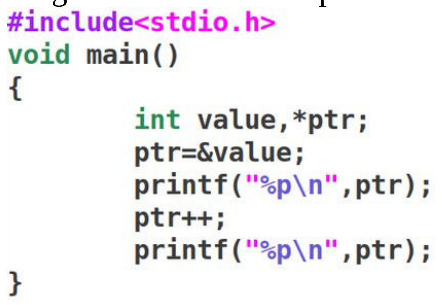
- As a pointer holds the memory address of a variable, some arithmetic operations can be performed with pointers.
- C supports 4 arithmetic operators that can be used with pointer, such as

Addition	+
Subtraction	-
Incrementation	++
Decrementation	--

- Pointers are variables. They are not integers, but they can be displayed as unsigned integers.
- The conversion specifier for a pointer is added and subtracted.
  For example,

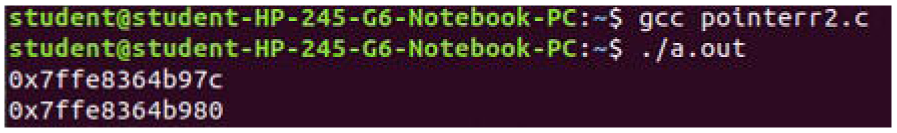
ptr++	causes the pointer to be incremented, but not by 1
ptr--	causes the pointer to be decremented, but not by 1.

The following program segment illustrates the pointer arithmetic.



```c
#include<stdio.h>
void main()
{
        int value,*ptr;
        ptr=&value;
        printf("%p\n",ptr);
        ptr++;
        printf("%p\n",ptr);
}
```

Output:



```
student@student-HP-245-G6-Notebook-PC:~$ gcc pointerr2.c
student@student-HP-245-G6-Notebook-PC:~$ ./a.out
0x7ffe8364b97c
0x7ffe8364b980
```

- The incrementation, ptr++, increments the number of bytes of the storage class for the particular machine.

- The general rule for pointer arithmetic is that pointer performs the operation in bytes of the appropriate storage class.

Pointer to Pointers in C

- In the c programming language, we have pointers to store the address of variables of any datatype.
- A pointer variable can store the address of a normal variable.
- C programming language also provides a pointer variable to store the address of another pointer variable.
- This type of pointer variable is called a pointer to pointer variable.
- Sometimes we also call it a double pointer.
- We use the following syntax for creating pointer to pointer…

```
datatype **pointerName ;
```

Example:

int **ptr;

Here, ptr is an integer pointer variable that stores the address of another integer pointer variable but does not stores the normal integer variable address.

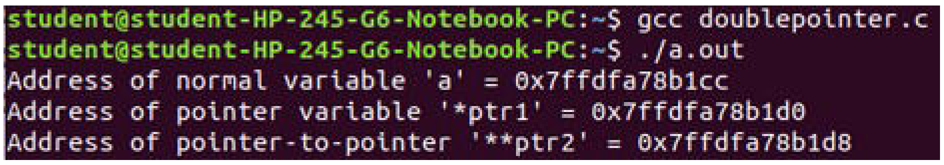
MOST IMPORTANT POINTS TO BE REMEMBERED

1. To store the address of normal variable we use single pointer variable
2. To store the address of single pointer variable we use double pointer variable
3. To store the address of double pointer variable we use triple pointer variable
4. Similarly the same for remaining pointer variables also…

Program

```
#include<stdio.h>

int main()
{
   int a ;
   int *ptr1 ;
   int **ptr2 ;
   int ***ptr3 ;

   ptr1 = &a ;
   ptr2 = &ptr1 ;
   ptr3 = &ptr2 ;

   printf("Address of normal variable 'a' = %p\n", ptr1) ;
   printf("Address of pointer variable '*ptr1' = %p\n", ptr2) ;
   printf("Address of pointer-to-pointer '**ptr2' = %p\n", ptr3) ;
    return 0;
}
```

Output:



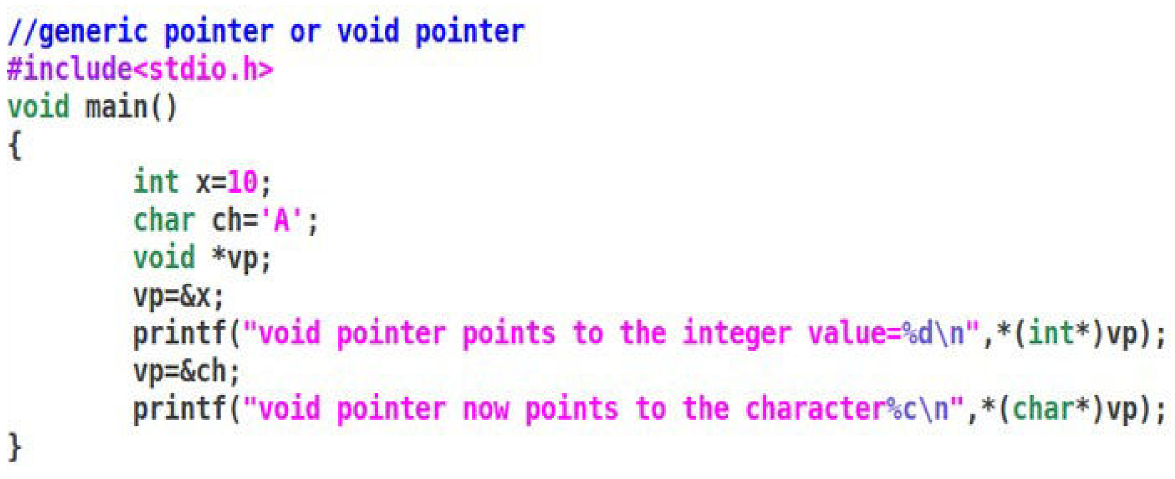
```
student@student-HP-245-G6-Notebook-PC:~$ gcc doublepointer.c
student@student-HP-245-G6-Notebook-PC:~$ ./a.out
Address of normal variable 'a' = 0x7ffdfa78b1cc
Address of pointer variable '*ptr1' = 0x7ffdfa78b1d0
Address of pointer-to-pointer '**ptr2' = 0x7ffdfa78b1d8
```

void pointer

- A void pointer is a pointer variable that has void as its data type.
- The void pointer is a special type of pointer that can be used to point to variables of any data type.
- It is declared like a normal pointer variable but using the void keyword as the pointer’s data type.
- For example

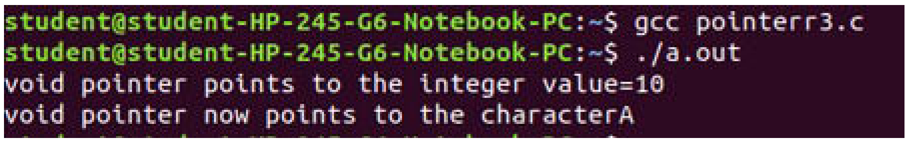
void *ptr;

Example:



```
//generic pointer or void pointer
#include<stdio.h>
void main()
{
        int x=10;
        char ch='A';
        void *vp;
        vp=&x;
        printf("void pointer points to the integer value=%d\n",*(int*)vp);
        vp=&ch;
        printf("void pointer now points to the character%c\n",*(char*)vp);
}
```

Output:



```
student@student-HP-245-G6-Notebook-PC:~$ gcc pointerr3.c
student@student-HP-245-G6-Notebook-PC:~$ ./a.out
void pointer points to the integer value=10
void pointer now points to the characterA
```

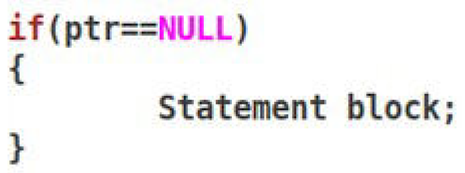
NULL POINTER

- We have seen that pointers variable is a pointer to some other variable of the same data type.
- However in some cases we may prefer to have ***null pointer,*** which is a special pointer value that does not point anywhere.
- This means that
  a NULL pointer is a pointer that does not point to any valid memory address.
- To declare a null pointer we may use the predefined constant NULL.

  Ex:

```
int *ptr=NULL;
```

- We can always check whether a given pointer variable stores address of some variable or contains a NULL by writing



```
if(ptr==NULL)
{
        Statement block;
}
```

- We may also initialize a pointer as a null pointer by using a constant 0, as follows:

```
int *ptr;
ptr=0;
```

This is a valid statement in C.

DANGLING POINTER

- **Dangling pointer is a pointer which points to some non-existing memory location.**

- Dangling pointers arise when an object is deleted or deallocated, without modyfying the value of the pointer.

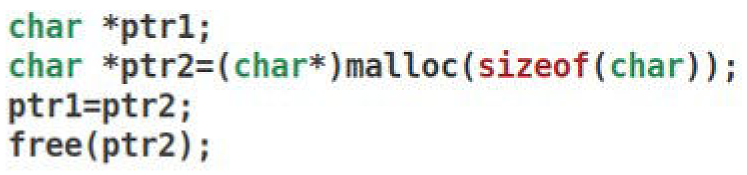
- As a result, the pointer still points to the memory location of the de-allocated memory.

- Once the memory is de-allocated, the system may reallocate that block of freed memory to another process.

- In case the program then dereferences the (now) dangling pointer, *unpredictable behaviour may result*, as the memory may now contain completely different data.

- Hence dangling pointer problem occurs when the pointer still points to the same location in memory even though the reference has been deleted and may now be used for other purpose.

- Consider the following code which illustrates dangling pointer problem



```
char *ptr1;
char *ptr2=(char*)malloc(sizeof(char));
ptr1=ptr2;
free(ptr2);
```

- Now ptr1 becomes a dangling pointer

const qualifier

If any variable is declared with **const** qualifier, then the program can't change the value of that variable. The const qualifier can occur in the declaration before or after the data type.

For example:

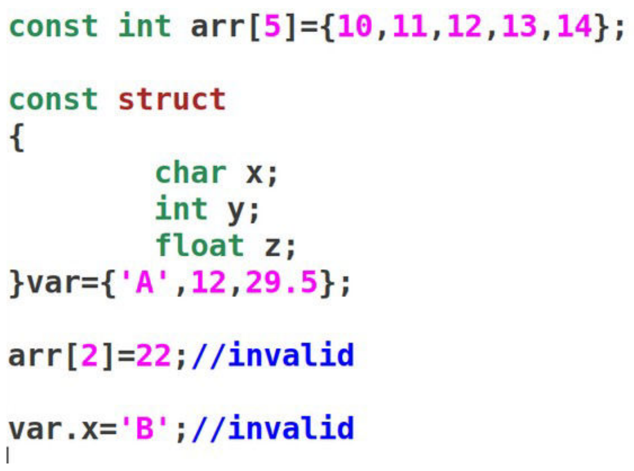
```
const int x=9;
int const x=9;
```

Both these declarations are equivalent and declare x as a const varaible. Any attempt to change value of this variable in the program will result an error. For example these statements are invalid-

```
x=10;        //invalid
x=func();    //invalid
x++;         //invalid
```

The const qualifier informs the compiler that the variable cam be stored in read only memory. A const variable can be given a value only through initialization or by some hardware devices. Note that the program can't modify the value of a const varaible, but any external event outside the program can change its value.

If an array, structure or union is declared as const then each member in it becomes constant. For example-

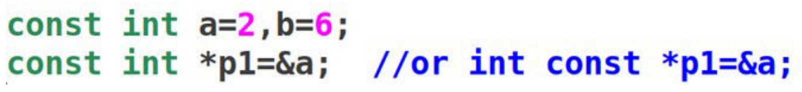


```
const int arr[5]={10,11,12,13,14};

const struct
{
        char x;
        int y;
        float z;
}var={'A',12,29.5};

arr[2]=22;//invalid

var.x='B';//invalid
```

Now we 'll see how to use const qualifier in pointer delcarations. We can declare three types of pointer using qualifier const

(1) Pointer to const data
(2) const pointer
(3) const pointer to const data

Consider these declarations-



```
const int a=2,b=6;
const int *p1=&a;   //or int const *p1=&a;
```

Here p1 is declared as a pointer to const integer. We can change the pointer p1 but we can't change the variable pointed to by p1.

```
*p1=9;    //invalid
p1=&b;  //valid since p1 is not a constant itself
```

Now consider these declarations-

```
int a=2,b=6;
int *const p2=&a;
```

Here p2 is declared as a const pointer. We cant change the pointer variable p2, but we can change the variable pointed to by p2.

```
*p2=9;    //valid
p2=&b;  //invalid since p2 is a constant
```

Now consider these declarations-

```
const int a=2,b=6;
const int const *p3=&a;
```

Here p3 is declared as a const pointer to const integer. We can neither change the pointer variable p3 nor the variable pointed to by it.

```
*p3=9;  //invalid
p3=&b   //invalid
```

So the three different types of declarations of pointers using const are-

```
int const *ptr; or      const int *ptr; //pointer to const integer
int *const ptr  or      //const pointer to an integer
const int *const ptr;   //const pointer to a const integer
```

Array and Pointers

- An array is a collection of variables of the same datatype.
- When an array is declared, the compiler allocates a base address and sufficient amount of storage to contain all the lements of the array in contiguous memory locations.
- The base address is the location of the first element (index 0) of the array.

Example

```
int arr[5]={10,20,30,40,50};
```

where,

arr will point the first element in the array which is 10.

Hence, arr will have the address of arr[0] (**&arr[0]**).

arr is exactly as same as &arr[0]. **arr == &arr[0]**

arr+i

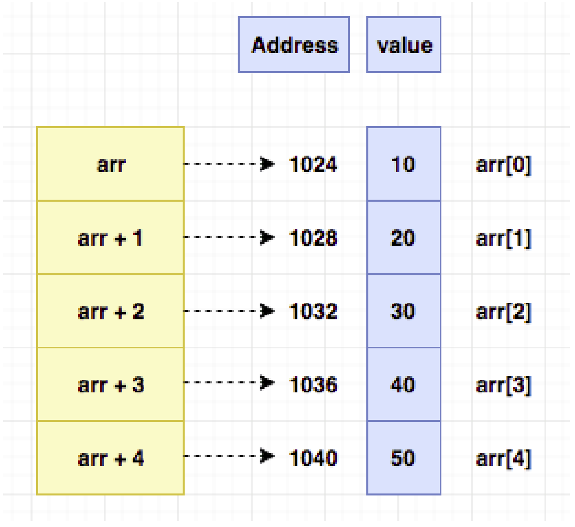
Assume that the first element address as 1024. So, **arr** will point the memory address 1024.

What will happen if we move arr by 1 position? i.e. arr+1?

It will not add 1 to 1024 instead it will move 4 bytes of memory as the size of an integer is 4 bytes.

So, arr + 1 will be 1028.

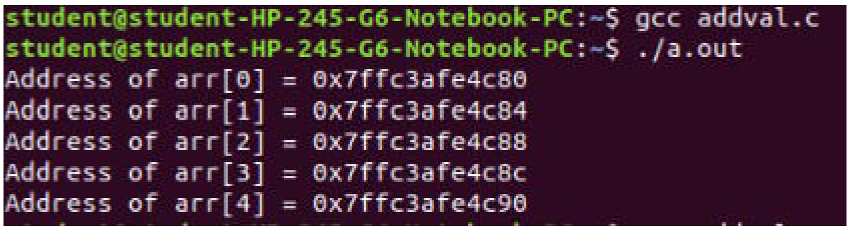
arr + 1 will point the second element of an array.



Program for printing array elements address

```c
#include<stdio.h>
int main()
{
    int arr[5]={10,20,30,40,50};
    for(int i=0;i<5;i++)
        printf("Address of arr[%d]=%p\n",i,arr+i);
    return 0;
}
```

Output:



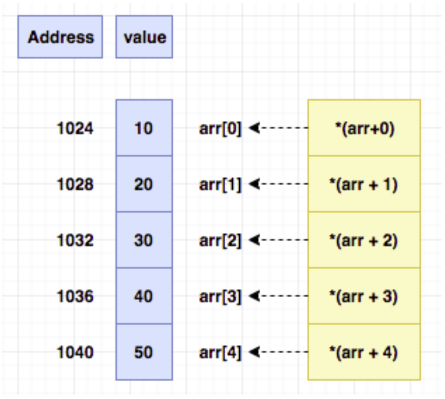
```
student@student-HP-245-G6-Notebook-PC:~$ gcc addval.c
student@student-HP-245-G6-Notebook-PC:~$ ./a.out
Address of arr[0] = 0x7ffc3afe4c80
Address of arr[1] = 0x7ffc3afe4c84
Address of arr[2] = 0x7ffc3afe4c88
Address of arr[3] = 0x7ffc3afe4c8c
Address of arr[4] = 0x7ffc3afe4c90
```

%p format specifier is used to printing the pointer address.

arr[i] == *(arr+i)

arr[i] is exactly same as *(arr+i) (value stored at the address arr+i).

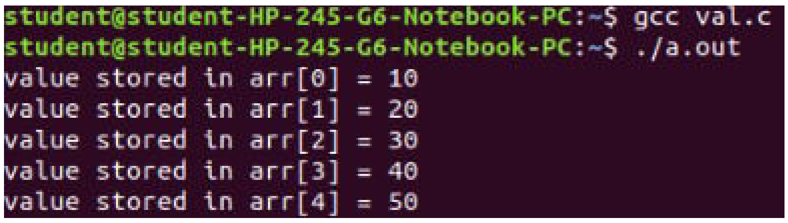
arr[i] will give the value stored at the memory address arr + i.



Program for printing elements in the array:

```c
#include<stdio.h>
int main()
{
    int arr[5]={10,20,30,40,50};
    for(int i=0;i<5;i++)
        printf("value stored in arr[%d] = %d\n",i,*(arr+i));
    return 0;
}
```

Output:



```
student@student-HP-245-G6-Notebook-PC:~$ gcc val.c
student@student-HP-245-G6-Notebook-PC:~$ ./a.out
value stored in arr[0] = 10
value stored in arr[1] = 20
value stored in arr[2] = 30
value stored in arr[3] = 40
value stored in arr[4] = 50
```

arr+1 vs &arr+1

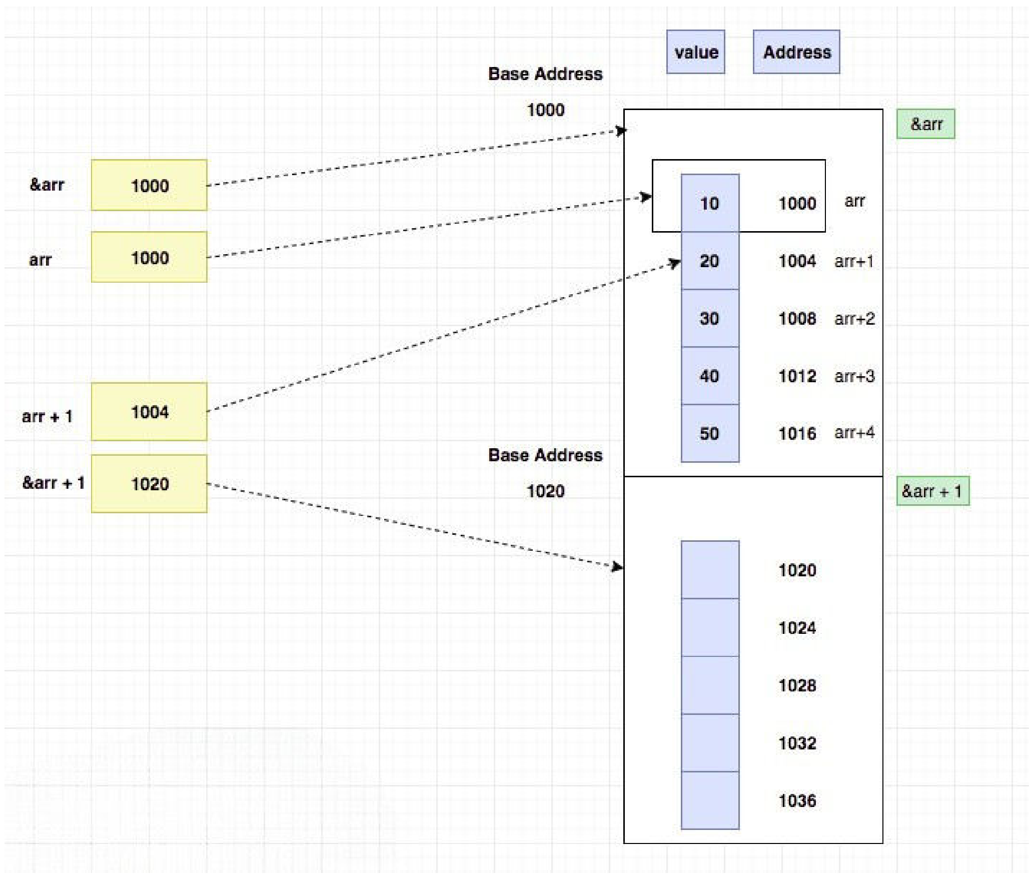
```
int arr[5] = {10,20,30,40,50};
```

Here,

arr is an integer pointer (int*) which points the first element of the array.

&arr is an integer array pointer (int*)[5] which points the whole array. (all five elements.)

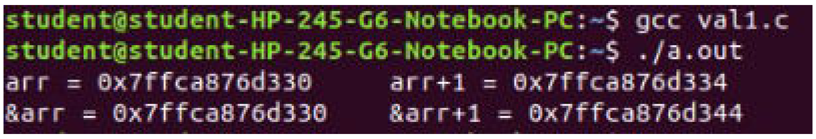
&arr+1	**arr + 1**
&arr is a pointer to an entire array. So, if we move &arr by 1 position it will point the next block of 5 elements.	arr is a pointer to the first element of the array.So, if we move arr by 1 position it will point the second element.
If the array base address is 1000, &arr+1 will be 1000 + (5 * 4) which is 1020	If the array base address is 1000, arr+1 will be 1000 + 4 which is 1004



Example:

```
#include<stdio.h>
int main()
{
        int arr[5]={10,20,30,40,50};
        printf("arr = %p \t arr+1 = %p\n",arr,arr+1);
        printf("&arr = %p \t &arr+1 = %p\n",&arr,&arr+1);
        return 0;
}
```

Output:



```
student@student-HP-245-G6-Notebook-PC:~$ gcc val1.c
student@student-HP-245-G6-Notebook-PC:~$ ./a.out
arr = 0x7ffca876d330     arr+1 = 0x7ffca876d334
&arr = 0x7ffca876d330    &arr+1 = 0x7ffca876d344
```

Accessing array elements using pointers

we can access the array elements using pointers.

Example

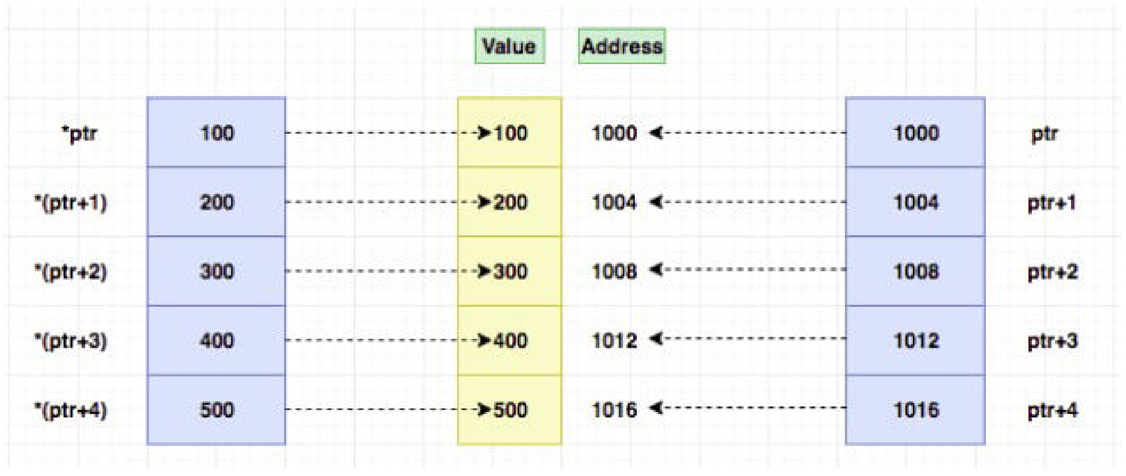
```
int arr[5] = {100, 200, 300, 400, 500};
int *ptr  = arr;
```

Where

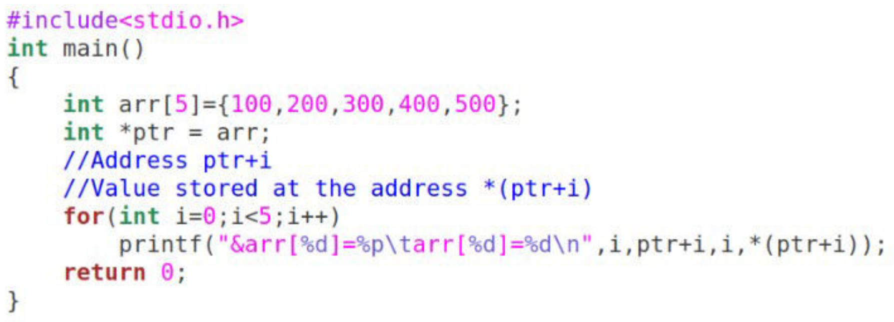
ptr is an integer pointer which holds the address of the first element. i.e **&arr[0]**

ptr + 1 points the address of second variable. i.e **&arr[1]**

Similarly, **ptr + 2** holds the address of the third element of the array. i.e. **&arr[2]**

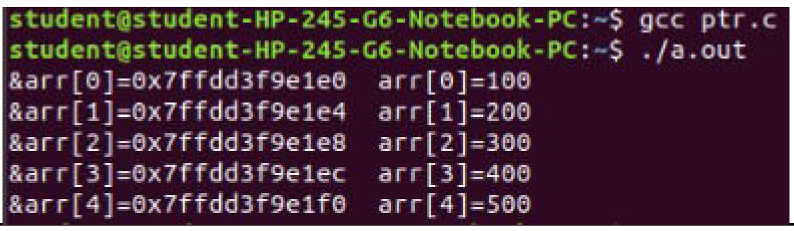


Printing each array elements address and value:



```
#include<stdio.h>
int main()
{
    int arr[5]={100,200,300,400,500};
    int *ptr = arr;
    //Address ptr+i
    //Value stored at the address *(ptr+i)
    for(int i=0;i<5;i++)
        printf("&arr[%d]=%p\tarr[%d]=%d\n",i,ptr+i,i,*(ptr+i));
    return 0;
}
```

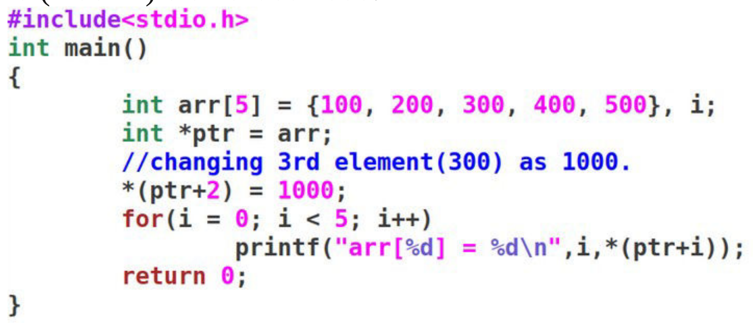
Output:



```
student@student-HP-245-G6-Notebook-PC:~$ gcc ptr.c
student@student-HP-245-G6-Notebook-PC:~$ ./a.out
&arr[0]=0x7ffdd3f9e1e0  arr[0]=100
&arr[1]=0x7ffdd3f9e1e4  arr[1]=200
&arr[2]=0x7ffdd3f9e1e8  arr[2]=300
&arr[3]=0x7ffdd3f9e1ec  arr[3]=400
&arr[4]=0x7ffdd3f9e1f0  arr[4]=500
```

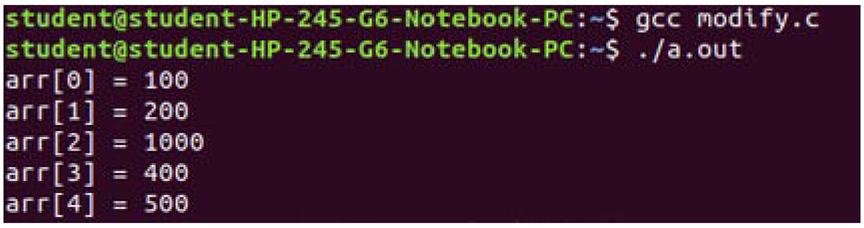
Modifying the array elements using the pointer:

Let's change the 3rd (index 2) element as 1000.



```c
#include<stdio.h>
int main()
{
        int arr[5] = {100, 200, 300, 400, 500}, i;
        int *ptr = arr;
        //changing 3rd element(300) as 1000.
        *(ptr+2) = 1000;
        for(i = 0; i < 5; i++)
                printf("arr[%d] = %d\n",i,*(ptr+i));
        return 0;
}
```

Output:



```
student@student-HP-245-G6-Notebook-PC:~$ gcc modify.c
student@student-HP-245-G6-Notebook-PC:~$ ./a.out
arr[0] = 100
arr[1] = 200
arr[2] = 1000
arr[3] = 400
arr[4] = 500
```

POINTER TO AN ARRAY

We can also point the whole array using pointers.

Using the array pointer, we can easily manipulate the multi-dimensional array.

Example

```
int arr[5] = {10, 20, 30, 40, 50};
int (*ptr)[5];
ptr = &arr;
```

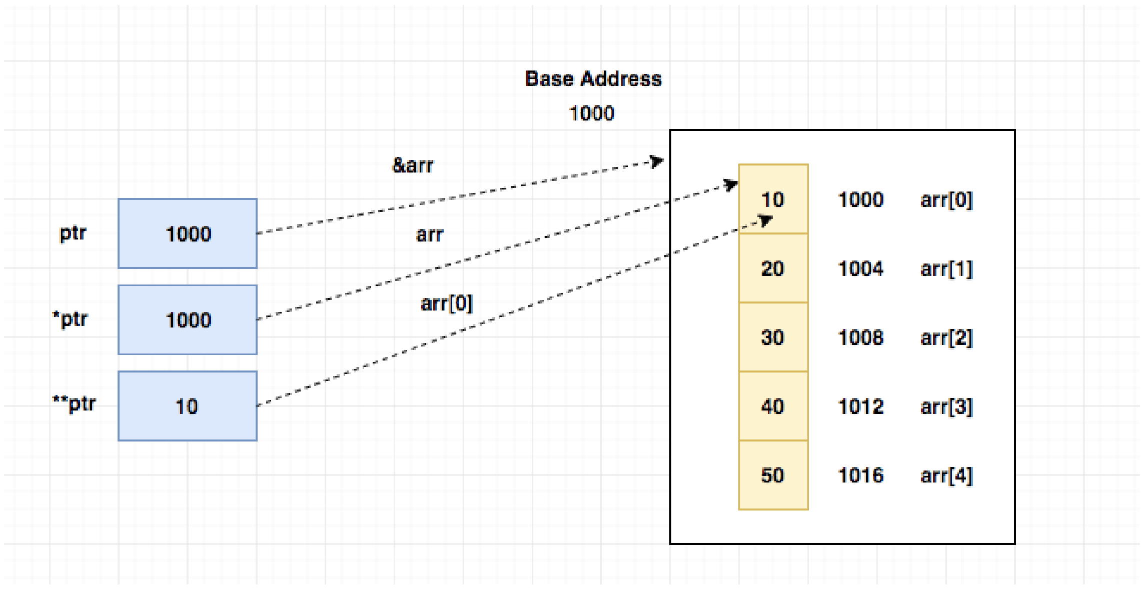
Where, **ptr** points the entire array.

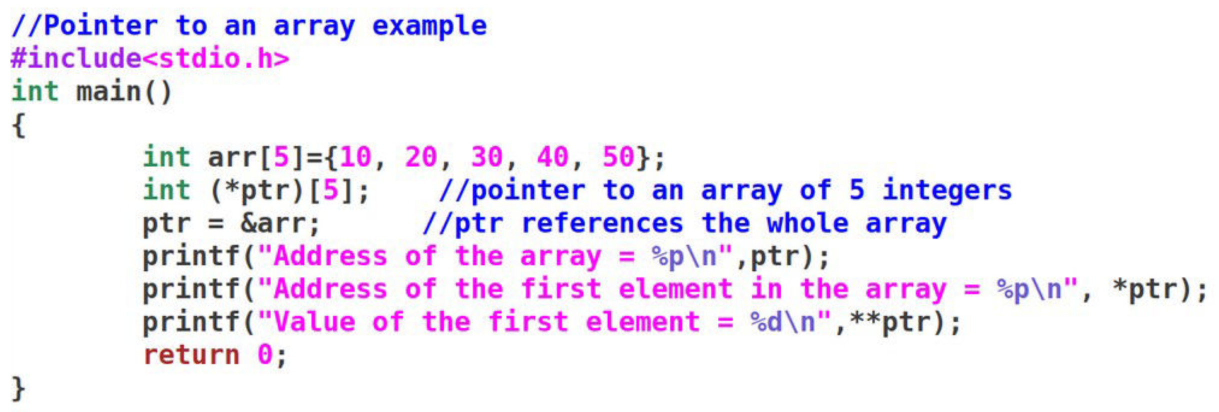
Dereferencing the array pointer:

Since ptr is an array pointer,

***ptr** will be again an address which is the address of the first element in the array.

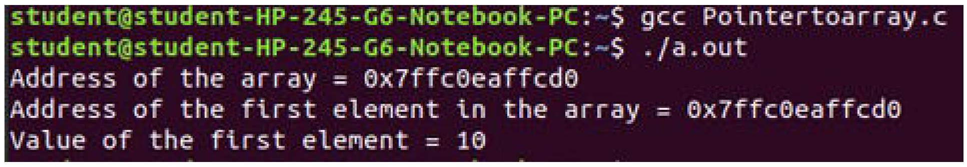
****ptr** will be the value stored at the address.





```
//Pointer to an array example
#include<stdio.h>
int main()
{
        int arr[5]={10, 20, 30, 40, 50};
        int (*ptr)[5];    //pointer to an array of 5 integers
        ptr = &arr;      //ptr references the whole array
        printf("Address of the array = %p\n",ptr);
        printf("Address of the first element in the array = %p\n", *ptr);
        printf("Value of the first element = %d\n",**ptr);
        return 0;
}
```

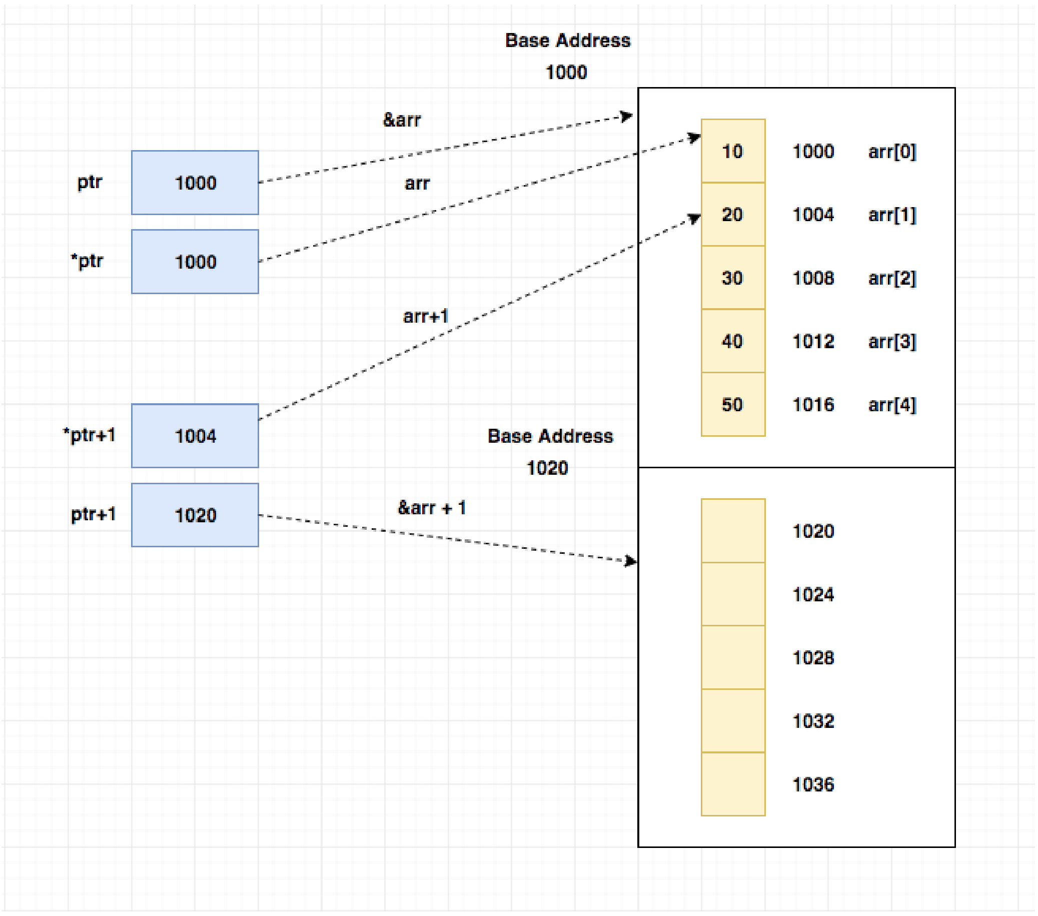
OUTPUT:



```
student@student-HP-245-G6-Notebook-PC:~$ gcc Pointertoarray.c
student@student-HP-245-G6-Notebook-PC:~$ ./a.out
Address of the array = 0x7ffc0eaffcd0
Address of the first element in the array = 0x7ffc0eaffcd0
Value of the first element = 10
```

ptr+1 vs *ptr+1:

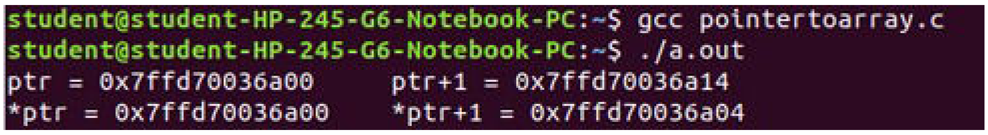
ptr + 1	*ptr + 1
ptr is a pointer to an entire array. So, if we move ptr by 1 position it will point the next block of 5 elements.	*ptr is a pointer to the first element of the array.So, if we move *ptr by 1 position it will point the second element.
If the array base address is 1000,ptr+1 will be 1000 + (5 * 4) which is 1020	If the array base address is 1000, *ptr+1 will be 1000 + 4 which is 1004



PROGRAM:

```
//Pointer to an array
#include<stdio.h>
int main()
{
        int arr[5]={10, 20, 30, 40, 50};
        int (*ptr)[5];    //pointer to an array of 5 integers
        ptr = &arr;      //ptr references the whole array
        //ptr is a pointer to the whole array
        //ptr+1 will point the next block of 5 elements
        printf("ptr = %p \t ptr+1 = %p\n",ptr,ptr+1);
        //ptr is a pointer to the first element in the array
        //ptr+1 will point the next element in the array
        printf("*ptr = %p \t *ptr+1 = %p\n",*ptr,*ptr+1);
        return 0;
}
```

OUTPUT:



```
student@student-HP-245-G6-Notebook-PC:~$ gcc pointertoarray.c
student@student-HP-245-G6-Notebook-PC:~$ ./a.out
ptr = 0x7ffd70036a00      ptr+1 = 0x7ffd70036a14
*ptr = 0x7ffd70036a00     *ptr+1 = 0x7ffd70036a04
```

Array of pointers in c

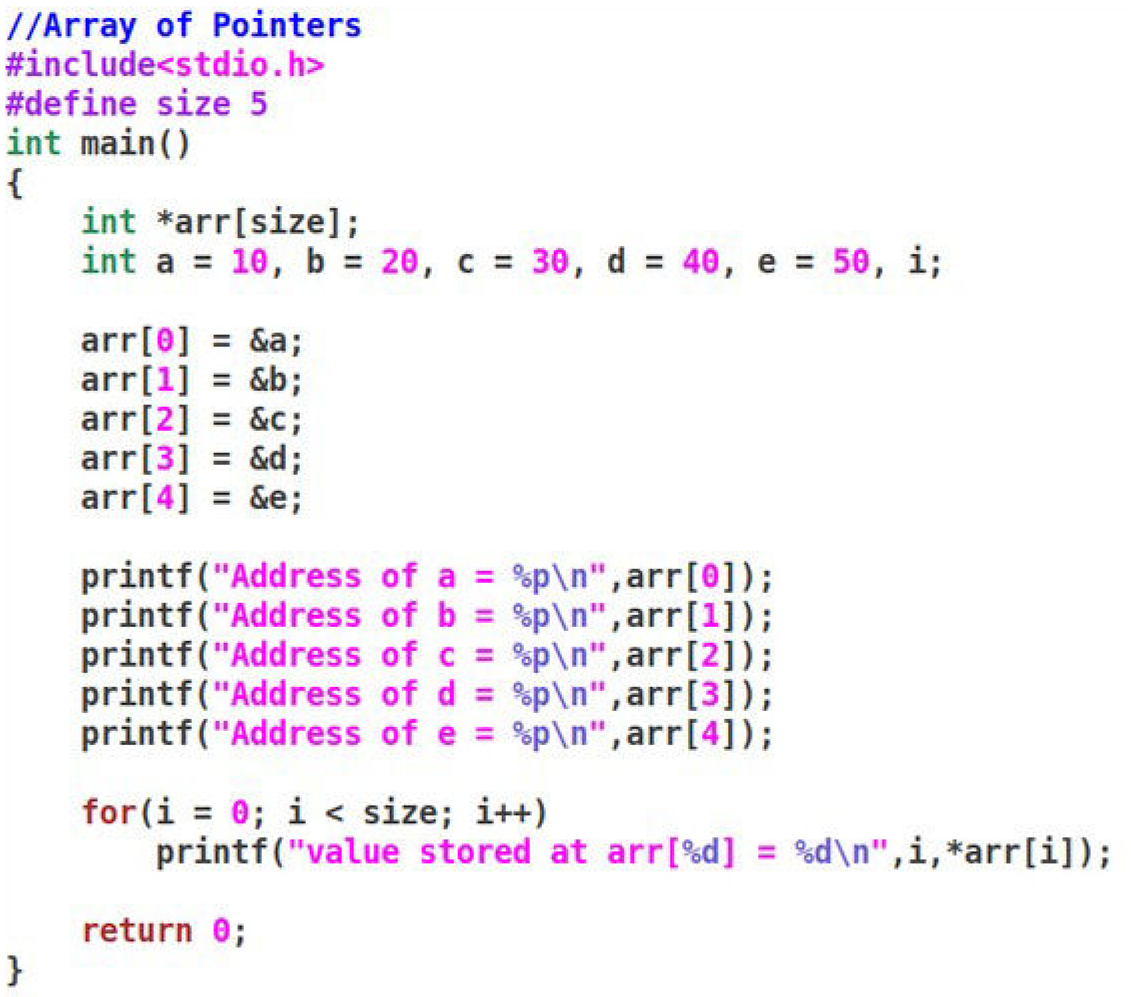
- Like an array of variables, we can also use array of pointers in c.

Array of Pointer:

Let's create an array of 5 pointers.

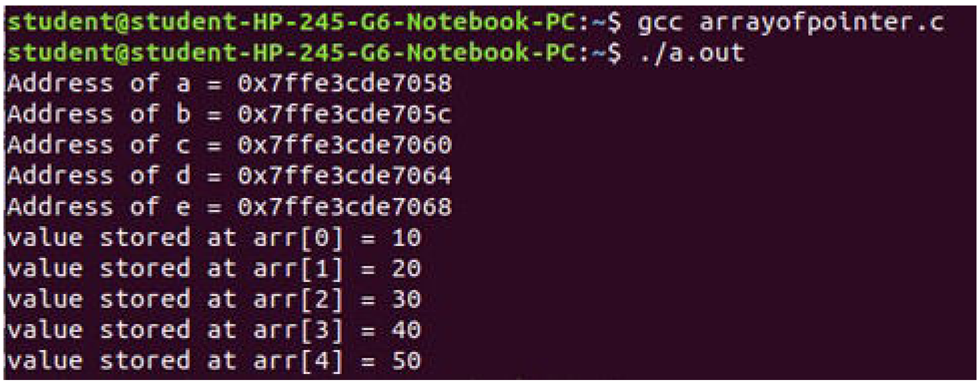
```
int *arr[5];
```

Where arr[0] will hold one integer variable address, arr[1] will hold another integer variable address and so on.



```
//Array of Pointers
#include<stdio.h>
#define size 5
int main()
{
    int *arr[size];
    int a = 10, b = 20, c = 30, d = 40, e = 50, i;

    arr[0] = &a;
    arr[1] = &b;
    arr[2] = &c;
    arr[3] = &d;
    arr[4] = &e;

    printf("Address of a = %p\n",arr[0]);
    printf("Address of b = %p\n",arr[1]);
    printf("Address of c = %p\n",arr[2]);
    printf("Address of d = %p\n",arr[3]);
    printf("Address of e = %p\n",arr[4]);

    for(i = 0; i < size; i++)
        printf("value stored at arr[%d] = %d\n",i,*arr[i]);

    return 0;
}
```

Output:



```
student@student-HP-245-G6-Notebook-PC:~$ gcc arrayofpointer.c
student@student-HP-245-G6-Notebook-PC:~$ ./a.out
Address of a = 0x7ffe3cde7058
Address of b = 0x7ffe3cde705c
Address of c = 0x7ffe3cde7060
Address of d = 0x7ffe3cde7064
Address of e = 0x7ffe3cde7068
value stored at arr[0] = 10
value stored at arr[1] = 20
value stored at arr[2] = 30
value stored at arr[3] = 40
value stored at arr[4] = 50
```

Dereferencing array of pointer:

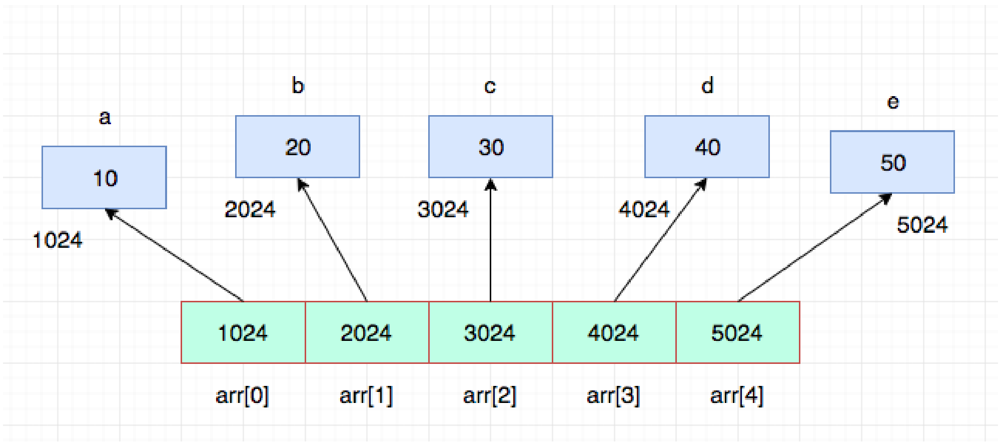
Dereference - Accessing the value stored at the pointer.

Since each array index pointing to a variable's address, we need to use *arr[index] to access the value stored at the particular index's address.

arr[index] will have address

*arr[index] will print the value.

Example



Where arr[0] holding the address of the variable a. i.e. arr[0] = 1024

*arr[0] -> *1024 -> value stored at the memory address 1024 is 10.

So, *arr[0] = 10.

Similarly, *arr[1] = 20, *arr[2] = 30 and so on.

Application of array of pointers:

Assume that we are going to build an embedded system which uses a different kind of sensors to measure the temperature.

In this case, we can use an array of pointers to hold the memory address of each sensor so that it will be very easy to manipulate the sensor status.

Example

sensor[0] will hold the address of the 1st sensor.

sensor[1] will hold the address of the second sensor and so on.

Since it is an array, we can directly interact with the particular sensor using array index.

we can get the temperature status of 1st sensor using sensor[0], 2nd sensor status using sensor[1] and so on.

2D array and pointers

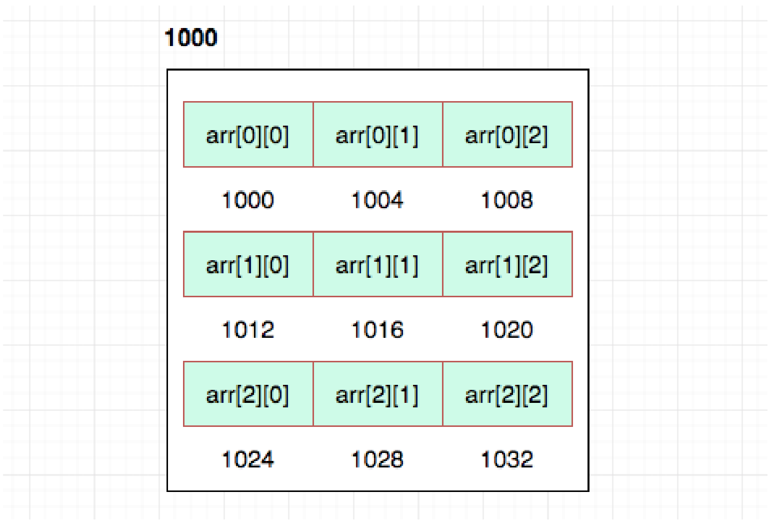
2D array

Let's declare a 3x3 array.

int arr[3][3];

Let's assume the starting address of the above 2D array as 1000.

Visual representation:



The above array's memory address is an arithmetic progression with common difference of 4 as the size of an integer is 4.

&Arr is a whole 2D array pointer

&arr is a pointer to the entire 2D(3x3) array. i.e. (int*)[3][3]

If we move &arr by 1 position(&arr+1), it will point to the next 2D block(3X3).

In our case, the base address of the 2D array is 1000. So, &arr value will be 1000.

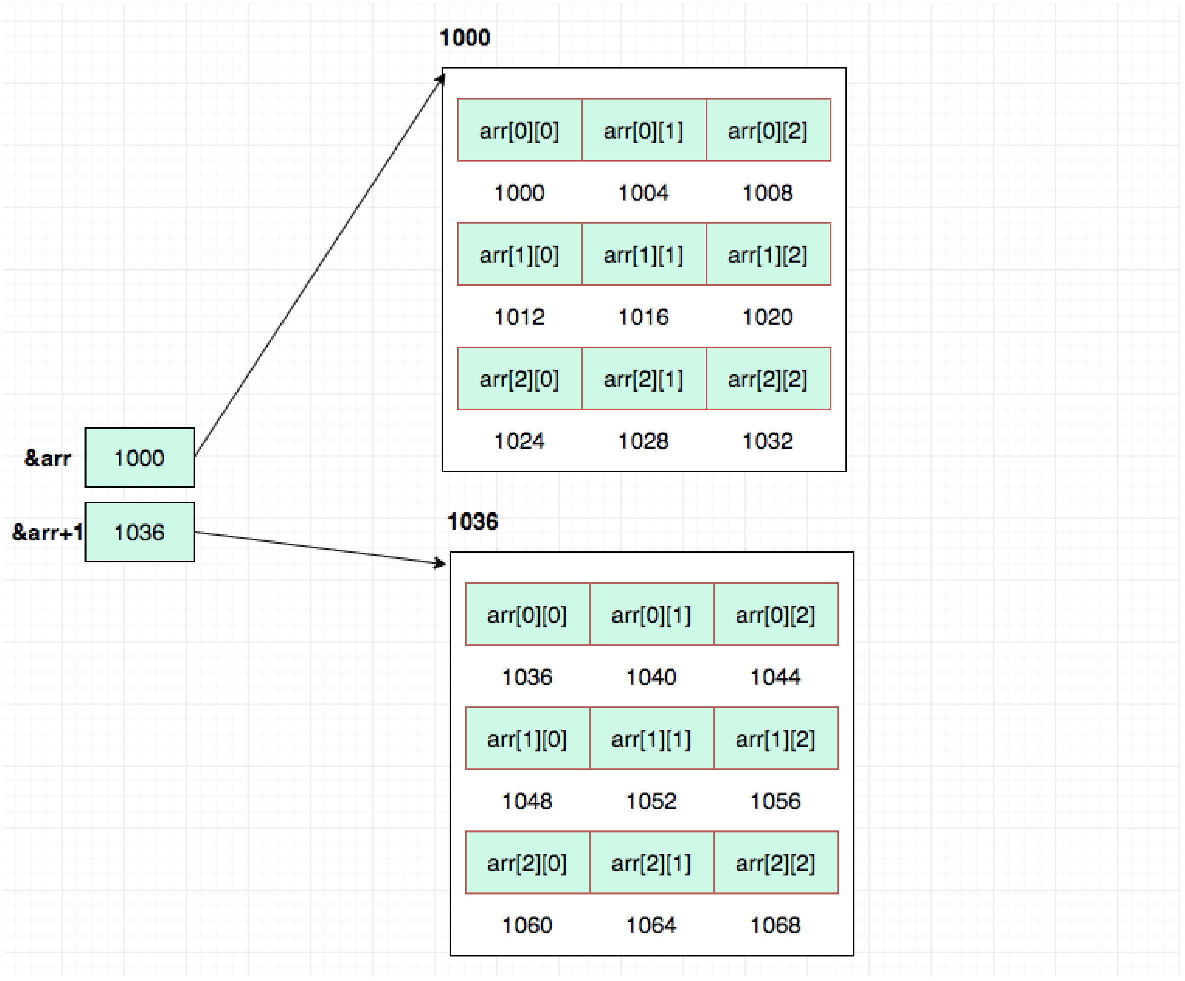
What will be the value of &arr+1?

In general, for Row x Col array the formula will be

base address + (Row * Col) * sizeof(array datatype)

In our case, Row = 3, Col = 3, sizeof(int) = 4.

It is base address + (3 * 3)* 4 => 1000 + 36 => 1036



Arr is a 1D array pointer

arr is a pointer to the first 1D array. i.e. (int*)[3]

If we move arr by 1 position(arr+1), it will point to the next 1D block(3 elements).

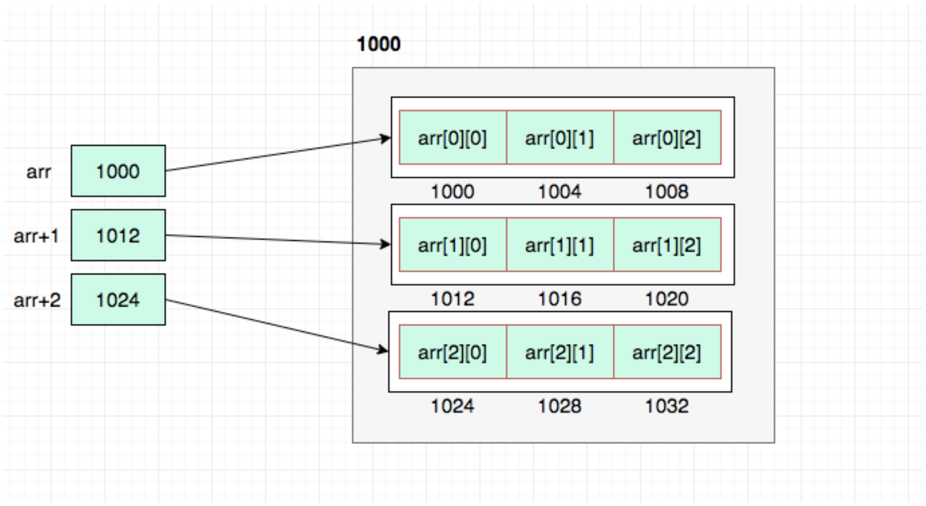
The base address of first 1D array also 1000. So, arr value will be 1000.

What will be the value of arr+1?

It will be base address + Row * sizeof(datatype).

1000 + 3 * 4

1012



*Arr is a pointer to the first element of the 2D array.

arr is a pointer to the first element of the 2D array. i.e. (int)

If we move *arr by 1 position(*arr+1), it will point to the next element.

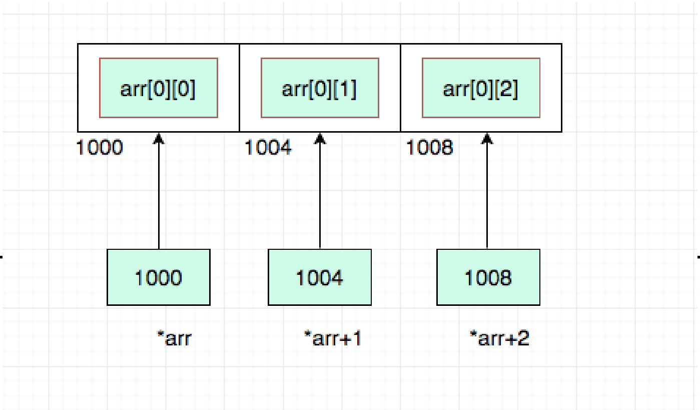
The base address of the first element also 1000.So, *arr value will be 1000.

What will be the value of *arr+1?

It will be base address + sizeof(datatype).

1000 + 4

1004



**arr will be the value of the first element.

Since *arr holds the address of the first element, **arr will give the value stored in the first element.

If we move **arr by 1 position(**arr+1), the value will be incremented by 1.

If the array first element is 10, **arr+1 will be 11.

Summary

1. &arr is a 2D array pointer (int*)[row][col]. So, &arr+1 will point the next 2D block.

2. arr is a 1D array pointer (int*)[row]. So, arr+1 will point the next 1D array in the 2D array.

3. *arr is a single element pointer (int*). So, *arr+1 will point the next element in the array.

4. **arr is the value of the first element. **arr+1 will increment the element value by 1.

String and Pointer

The string itself act as a pointer.

The string name is a pointer to the first character in the string.

String name == &string[0]

As we all know, the string is a character array which is terminated by the null '\0' character.

Example

```
char str[6]=”Hello”;
```

Where,

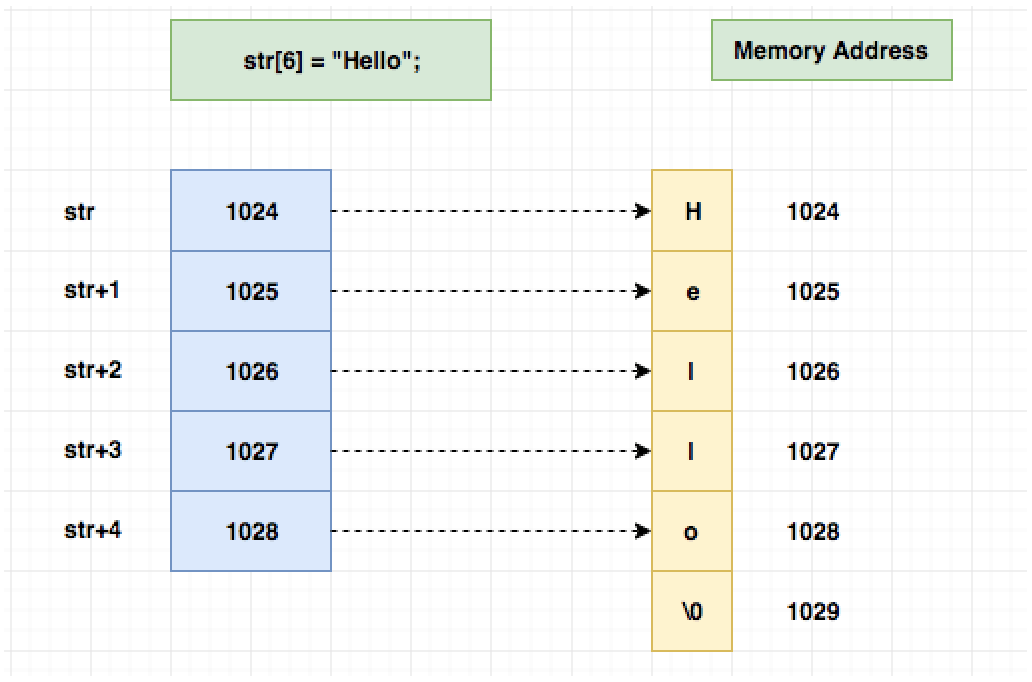
str is the string name and it is a pointer to the first character in the string. i.e. &str[0].

Similarly, str+1 holds the address of the second character of the string. i.e. &str[1]

To store "Hello", we need to allocate 6 bytes of memory.

5 byte for "Hello"

1 byte for null '\0' character.



Let's print each character address of the string 'str'

```c
#include<stdio.h>
int main()
{
    char str[6] = "Hello";
    int i;

    //printing each char address
    for(i = 0; str[i]; i++)
        printf("&str[%d] = %p\n",i,str+i);

    return 0;
}
```

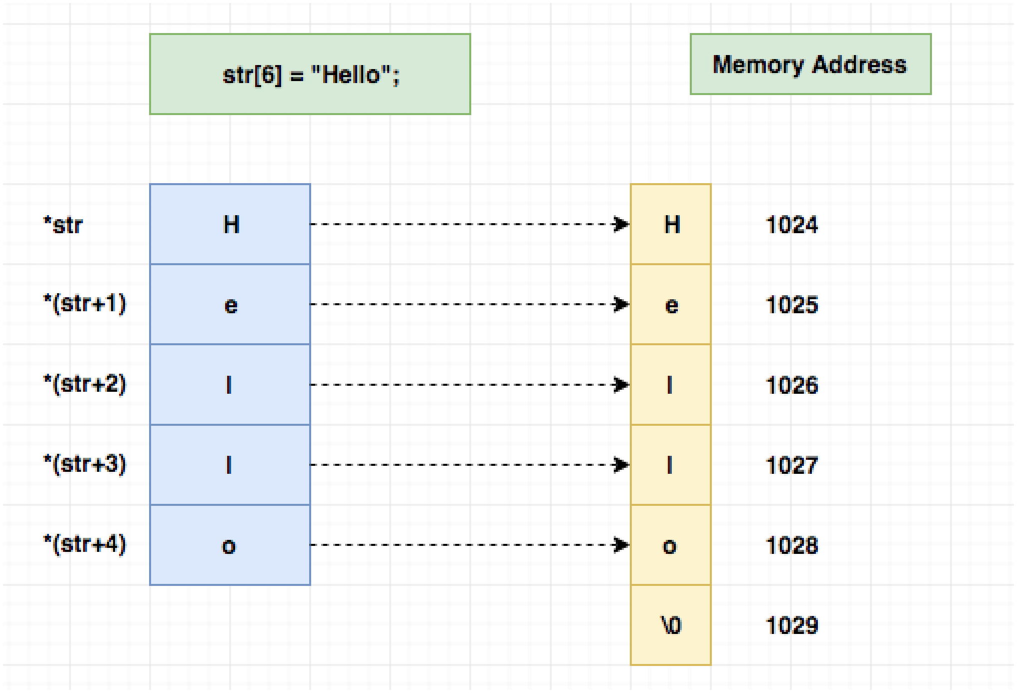
%p format specifier is used to printing the pointer address.

Str[i] == *(str+i)

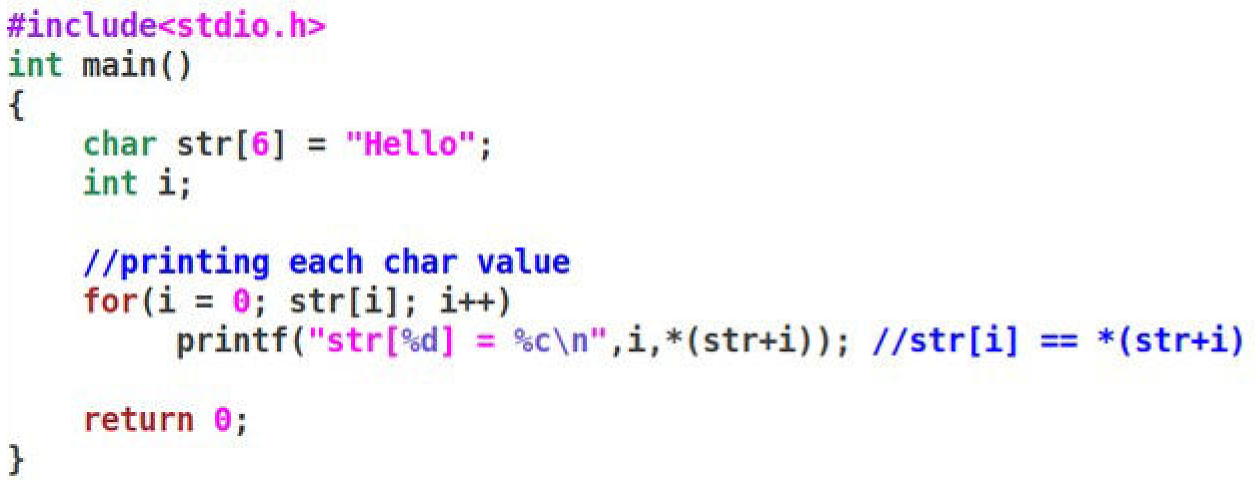
str[i] will give the character stored at the string index i.

str[i] is a shortend version of *(str+i).

*(str+i) - value stored at the memory address str+i. (base address + index)

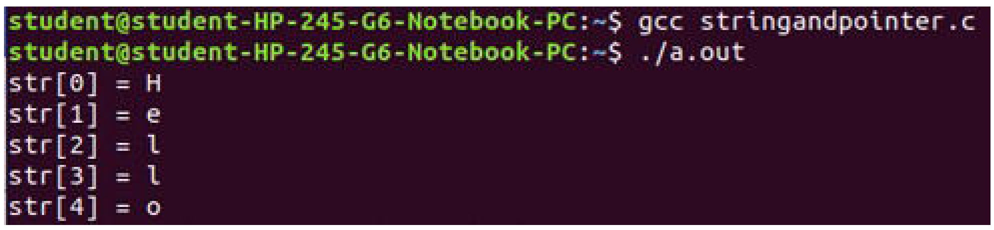


Let's print each character of a string using *(str+i)



```c
#include<stdio.h>
int main()
{
    char str[6] = "Hello";
    int i;

    //printing each char value
    for(i = 0; str[i]; i++)
        printf("str[%d] = %c\n",i,*(str+i)); //str[i] == *(str+i)

    return 0;
}
```

Output:



```
student@student-HP-245-G6-Notebook-PC:~$ gcc stringandpointer.c
student@student-HP-245-G6-Notebook-PC:~$ ./a.out
str[0] = H
str[1] = e
str[2] = l
str[3] = l
str[4] = o
```

Accessing string using pointer

Using char* (character pointer), we can access the string.

Example

Declare a char pointer

Assign the string base address(starting address) to the char pointer.

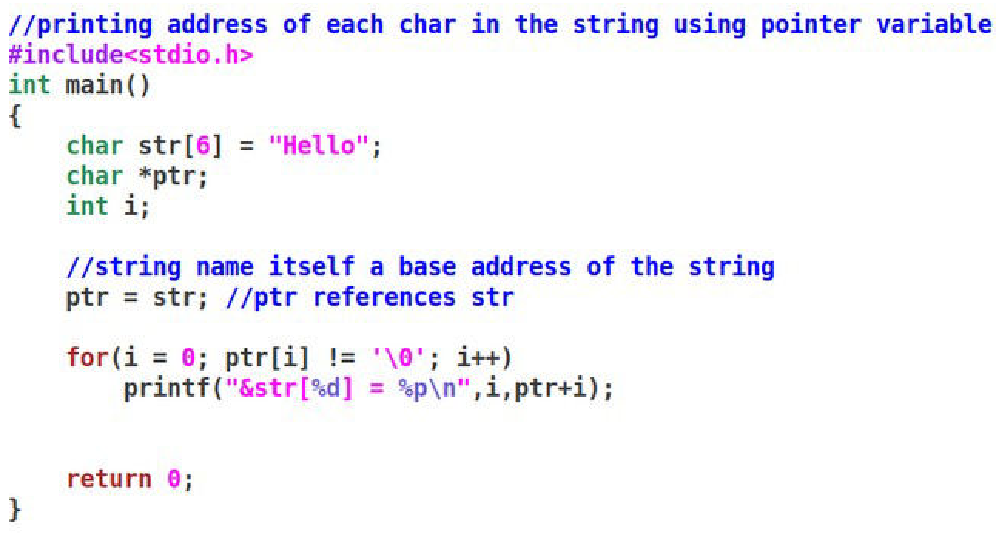
```
char str[6] = "Hello";
char *ptr;

//string name itself base address of the string
ptr = str; //ptr references str
```

Where,

ptr - is a character pointer which points the first character of the string. i.e. &str[0]

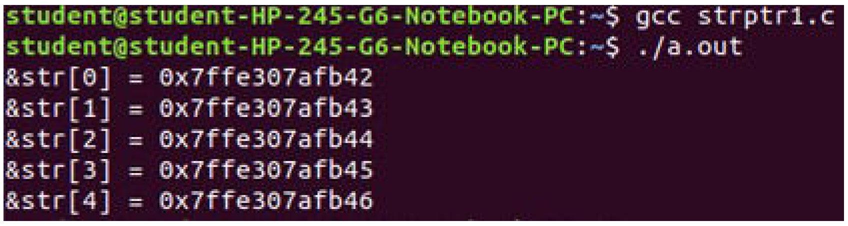
Like normal pointer arithmetic, if we move the ptr by 1 (ptr+1) position it will point the next character.

Printing each character address of a string using pointer variable



```
//printing address of each char in the string using pointer variable
#include<stdio.h>
int main()
{
    char str[6] = "Hello";
    char *ptr;
    int i;

    //string name itself a base address of the string
    ptr = str; //ptr references str

    for(i = 0; ptr[i] != '\0'; i++)
        printf("&str[%d] = %p\n",i,ptr+i);


    return 0;
}
```

Output:



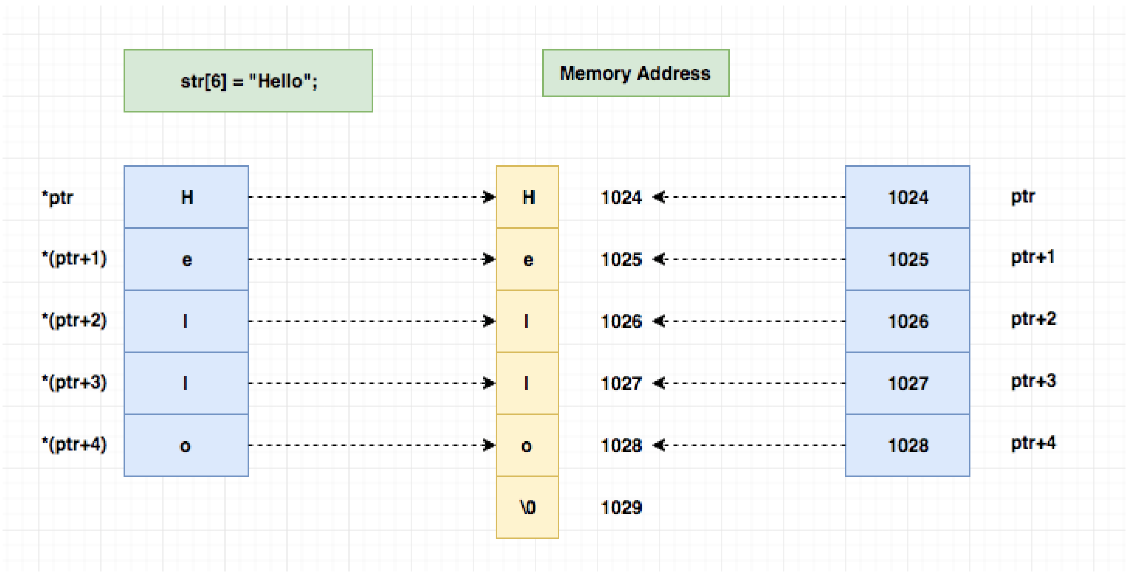
```
student@student-HP-245-G6-Notebook-PC:~$ gcc strptr1.c
student@student-HP-245-G6-Notebook-PC:~$ ./a.out
&str[0] = 0x7ffe307afb42
&str[1] = 0x7ffe307afb43
&str[2] = 0x7ffe307afb44
&str[3] = 0x7ffe307afb45
&str[4] = 0x7ffe307afb46
```

Printing each character of a string using the pointer

To print the value stored at each char address, we have to dereference the memory address.

Like below,

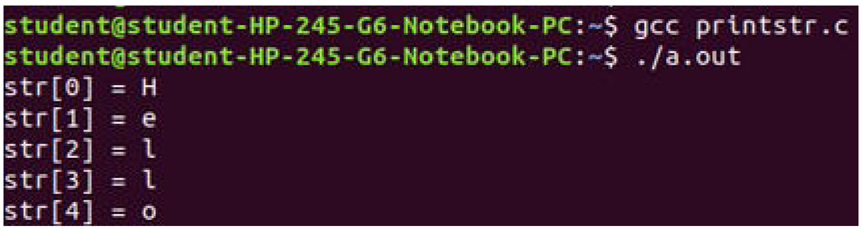
*(ptr+i)



Example:

```
//printing each char in the string using pointer variable
#include<stdio.h>
int main()
{
    char str[6] = "Hello";
    char *ptr;
    int i;

    //string name itself a base address of the string
    ptr = str; //ptr references str

    for(i = 0; ptr[i] != '\0'; i++)
        printf("str[%d] = %c\n",i,*(ptr+i));


    return 0;
}
```

Output:



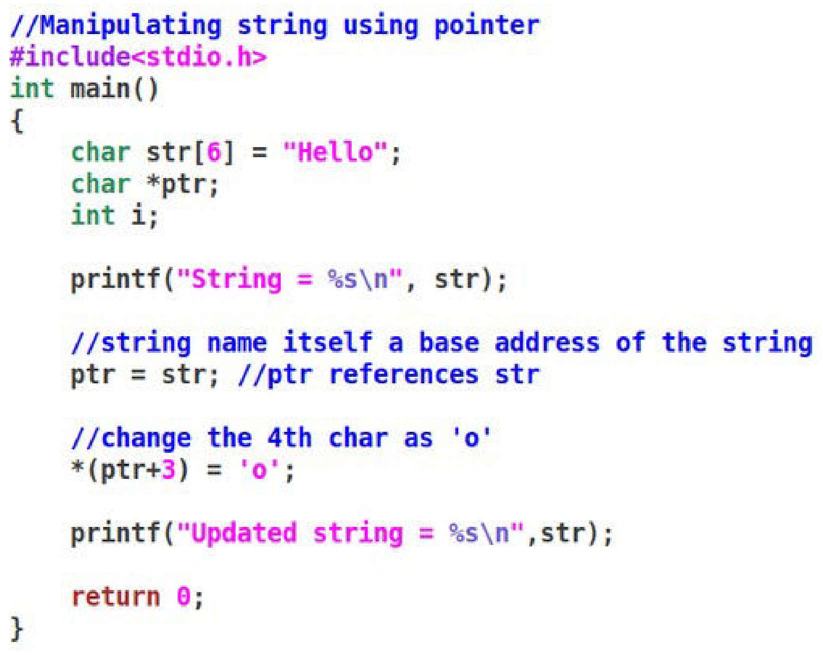
```
student@student-HP-245-G6-Notebook-PC:~$ gcc printstr.c
student@student-HP-245-G6-Notebook-PC:~$ ./a.out
str[0] = H
str[1] = e
str[2] = l
str[3] = l
str[4] = o
```

Manipulating characters of a string using the pointer

Let's change the 4th(index 3) character 'l' as 'o'.

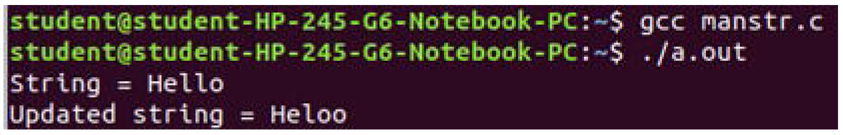
The new string will be "Heloo".

Example:



```
//Manipulating string using pointer
#include<stdio.h>
int main()
{
    char str[6] = "Hello";
    char *ptr;
    int i;

    printf("String = %s\n", str);

    //string name itself a base address of the string
    ptr = str; //ptr references str

    //change the 4th char as 'o'
    *(ptr+3) = 'o';

    printf("Updated string = %s\n",str);

    return 0;
}
```

Output:



```
student@student-HP-245-G6-Notebook-PC:~$ gcc manstr.c
student@student-HP-245-G6-Notebook-PC:~$ ./a.out
String = Hello
Updated string = Heloo
```

%S and string

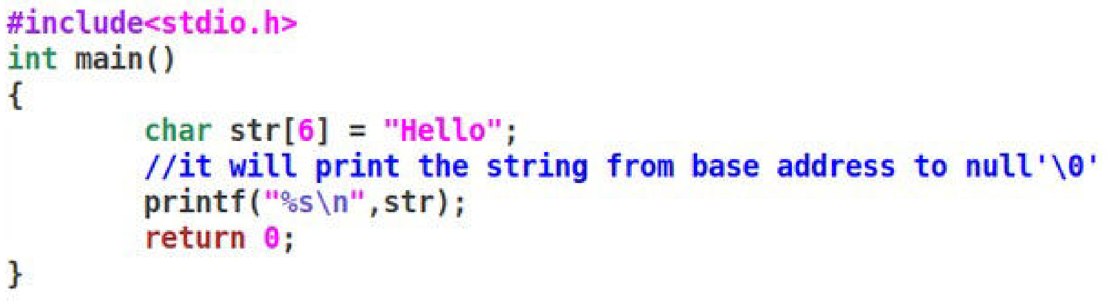
We can print the string using %s format specifier in printf function.

It will print the string from the given starting address to the null '\0' character.

String name itself the starting address of the string.

So, if we give string name it will print the entire string.

Printing string using %s format specifier



```
#include<stdio.h>
int main()
{
        char str[6] = "Hello";
        //it will print the string from base address to null'\0'
        printf("%s\n",str);
        return 0;
}
```

What will be the output if we give str+1?

If we give str+1, the string address moved by 1 byte (which is the address of the second character)

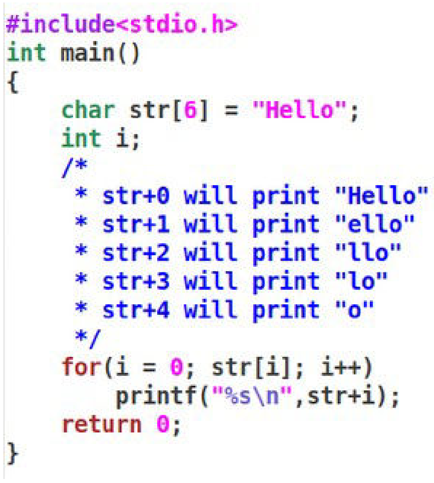
So, it will print the string from the second character.

Similarly, if we give str+2, it will print the string from the third character.

Lets print the following pattern:

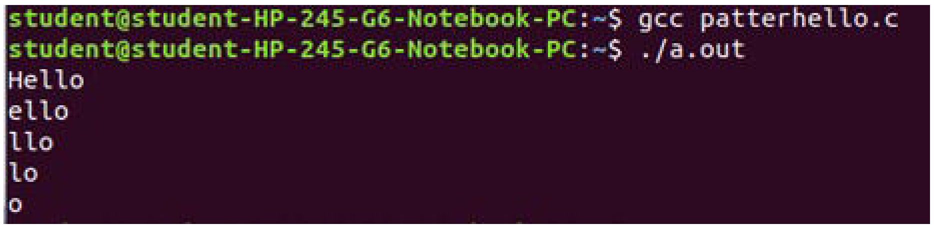
```
Hello
ello
llo
lo
o
```

Example:



```c
#include<stdio.h>
int main()
{
    char str[6] = "Hello";
    int i;
    /*
     * str+0 will print "Hello"
     * str+1 will print "ello"
     * str+2 will print "llo"
     * str+3 will print "lo"
     * str+4 will print "o"
     */
    for(i = 0; str[i]; i++)
        printf("%s\n",str+i);
    return 0;
}
```

Output:



```
student@student-HP-245-G6-Notebook-PC:~$ gcc patterhello.c
student@student-HP-245-G6-Notebook-PC:~$ ./a.out
Hello
ello
llo
lo
o
```

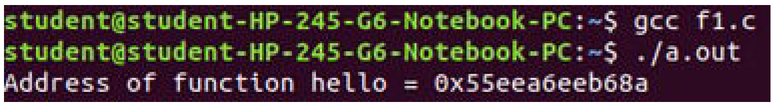
POINTER TO FUNCTION

- Like the pointer to normal variables, we can also use pointers to point the functions.
- Like every variables, function also has unique memory address.

PROGRAM:

```c
//Printing function's memory address
#include<stdio.h>
void hello()
{
        printf("Hello\n");
}
int main()
{
        //function name itself holds the memory address of a function
        printf("Address of function hello = %p\n",hello);
        return 0;
}
```

Output:



```
student@student-HP-245-G6-Notebook-PC:~$ gcc f1.c
student@student-HP-245-G6-Notebook-PC:~$ ./a.out
Address of function hello = 0x55eea6eeb68a
```

Function pointer syntax:

```c
void (*fptr)();
```

We must enclose the function pointer variable with (). like (*fptr).

Where ***fptr** is a function pointer, capable of referring a function whose return type is **void** and also **it will not take any argument ()**.

Similarly, if we want to point a function whose return type is **int** and it takes one integer argument i.e. int fun(int); then the declaration will be,

```c
int (*fptr)(int);
```

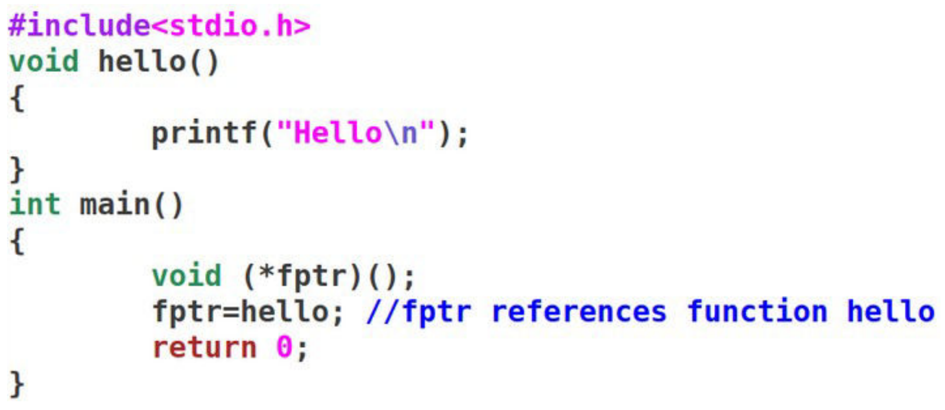
Referencing and Dereferencing of Function Pointer:

Referencing

1. Declare correct function pointer.

2. Assign function address to that pointer.

Example



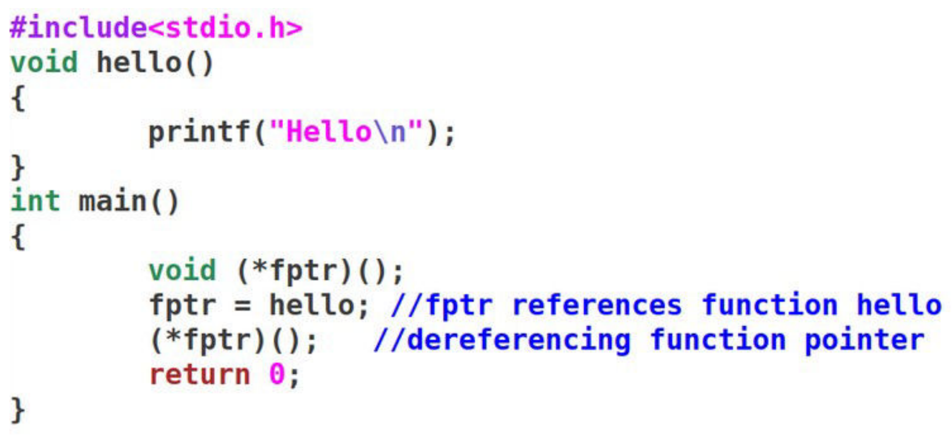
```
#include<stdio.h>
void hello()
{
        printf("Hello\n");
}
int main()
{
        void (*fptr)();
        fptr=hello; //fptr references function hello
        return 0;
}
```

Dereferencing

1. Calling the actual function using the function pointer.

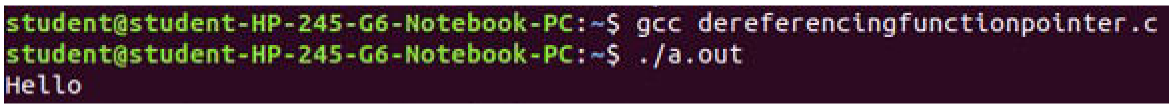
Using value at operator *, we can call the function. i.e. (*fptr)();

Example:



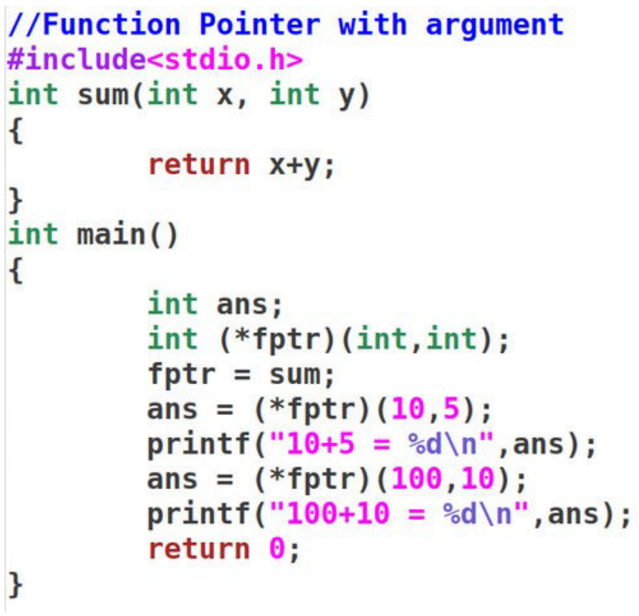
```
#include<stdio.h>
void hello()
{
        printf("Hello\n");
}
int main()
{
        void (*fptr)();
        fptr = hello; //fptr references function hello
        (*fptr)();    //dereferencing function pointer
        return 0;
}
```

Output:



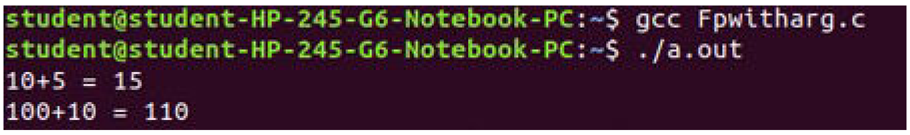
```
student@student-HP-245-G6-Notebook-PC:~$ gcc dereferencingfunctionpointer.c
student@student-HP-245-G6-Notebook-PC:~$ ./a.out
Hello
```

Function Pointer with Arguments:



```c
//Function Pointer with argument
#include<stdio.h>
int sum(int x, int y)
{
        return x+y;
}
int main()
{
        int ans;
        int (*fptr)(int,int);
        fptr = sum;
        ans = (*fptr)(10,5);
        printf("10+5 = %d\n",ans);
        ans = (*fptr)(100,10);
        printf("100+10 = %d\n",ans);
        return 0;
}
```

Output:



```
student@student-HP-245-G6-Notebook-PC:~$ gcc Fpwitharg.c
student@student-HP-245-G6-Notebook-PC:~$ ./a.out
10+5 = 15
100+10 = 110
```

PASSING POINTER TO FUNCTION

Pointer argument to function:

We can also pass the pointer variable to function.

In other words, we can pass the address of a variable to the function instead of variable value.

How to declare a function which accepts a pointer as an argument:

If a function wants to accept an address of an integer variable then the function declaration will be,

return_type **function_name**(**int***);

* indicates that the argument is an address of a variable not the *value of a variable*.

Similarly, If a function wants to accept an address of two integer variable then the function declaration will be,

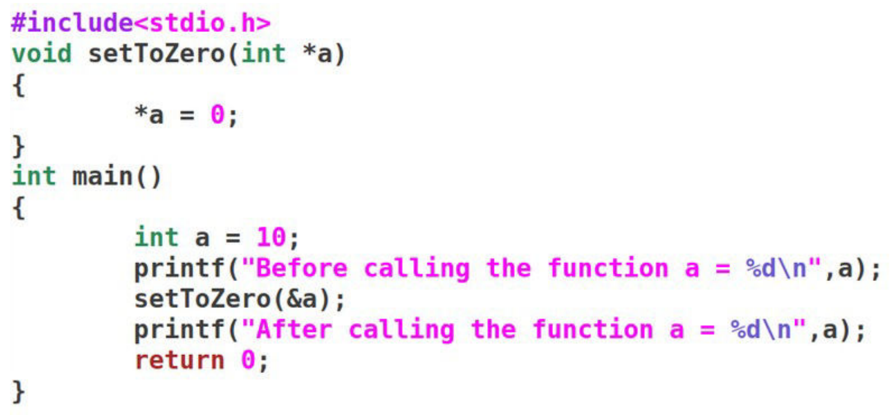
return_type **function_name**(**int***,**int***);

What will happen to the argument?

Here, we are passing the address of a variable.

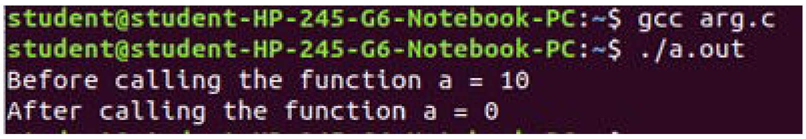
So, if we change the argument value in the function, it will modify the actual value of a variable.

Example:



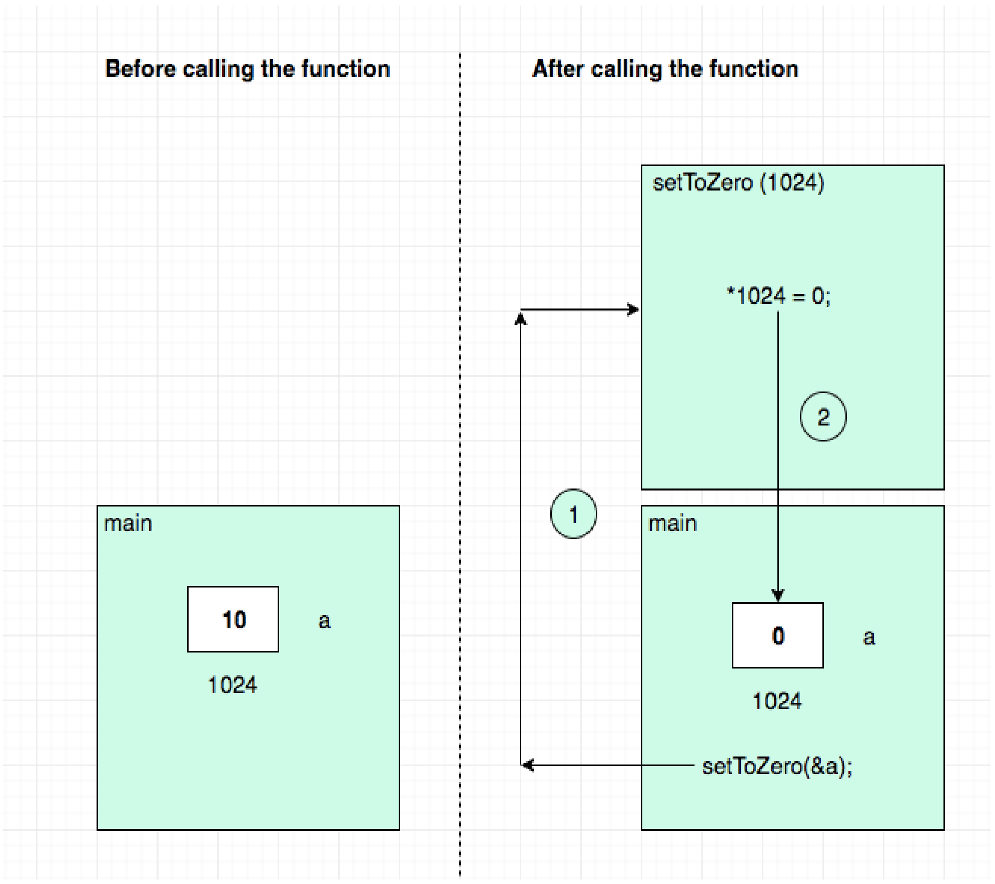
```c
#include<stdio.h>
void setToZero(int *a)
{
        *a = 0;
}
int main()
{
        int a = 10;
        printf("Before calling the function a = %d\n",a);
        setToZero(&a);
        printf("After calling the function a = %d\n",a);
        return 0;
}
```

Output:



```
student@student-HP-245-G6-Notebook-PC:~$ gcc arg.c
student@student-HP-245-G6-Notebook-PC:~$ ./a.out
Before calling the function a = 10
After calling the function a = 0
```

Visual Representation:



step 1. Address of a variable **a** 1024 is passed to function setToZero().

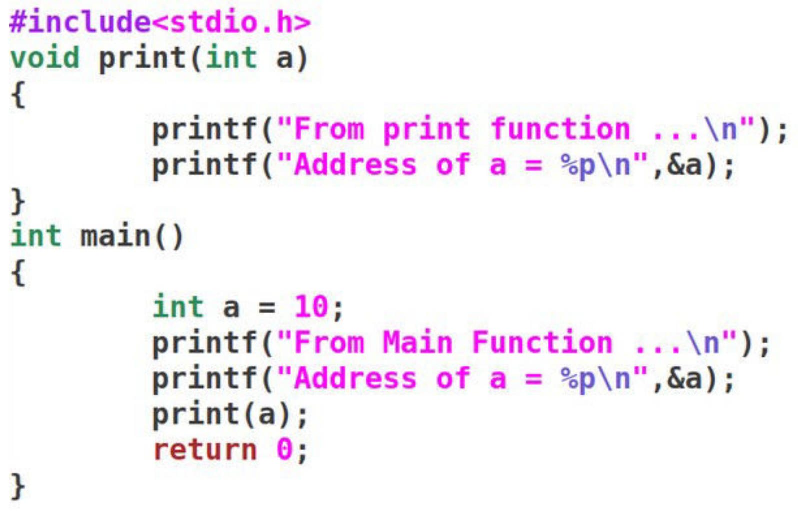
step 2. *a = 0 will modify the original value of **a**. So, a value in main function will be 0 after the function execution.

CALL BY VALUE

Call by value:

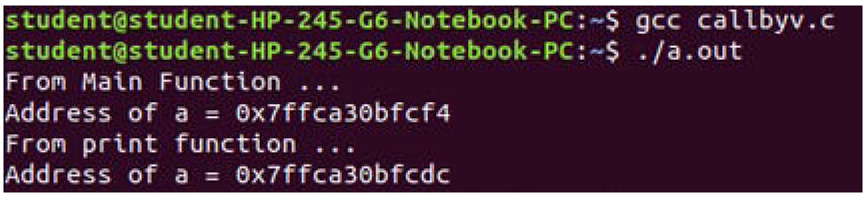
If we call a function with value (variable name), the value will be duplicated and the function will receive it.

Example:



```c
#include<stdio.h>
void print(int a)
{
	printf("From print function ...\n");
	printf("Address of a = %p\n",&a);
}
int main()
{
	int a = 10;
	printf("From Main Function ...\n");
	printf("Address of a = %p\n",&a);
	print(a);
	return 0;
}
```

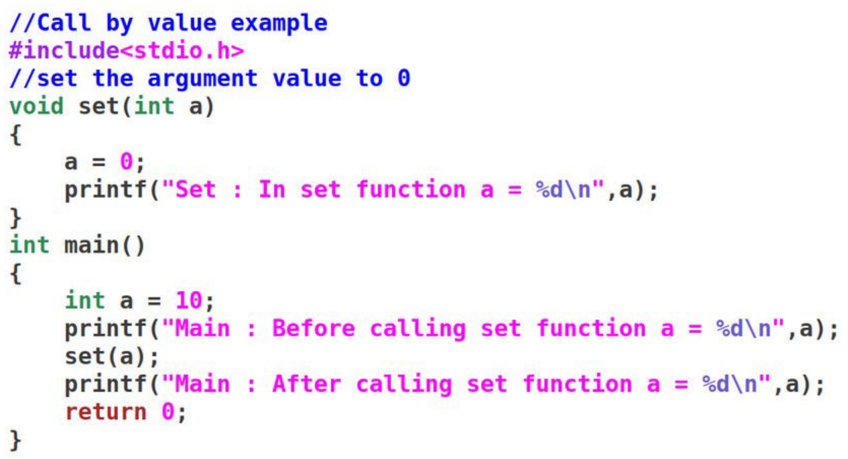
Output:



```
student@student-HP-245-G6-Notebook-PC:~$ gcc callbyv.c
student@student-HP-245-G6-Notebook-PC:~$ ./a.out
From Main Function ...
Address of a = 0x7ffca30bfcf4
From print function ...
Address of a = 0x7ffca30bfcdc
```

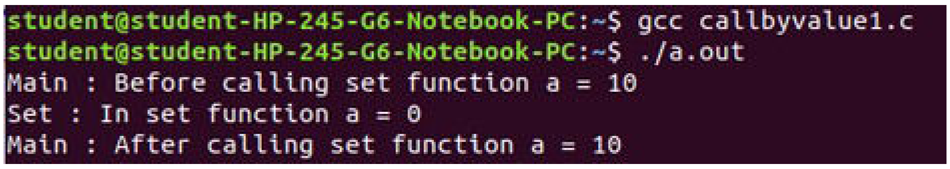
We can observe that the address of **a** in the main function and the address of **a** in print function are different.

So, if we change the value of **a** in print function, it will not affect the value of **a** in the main function.

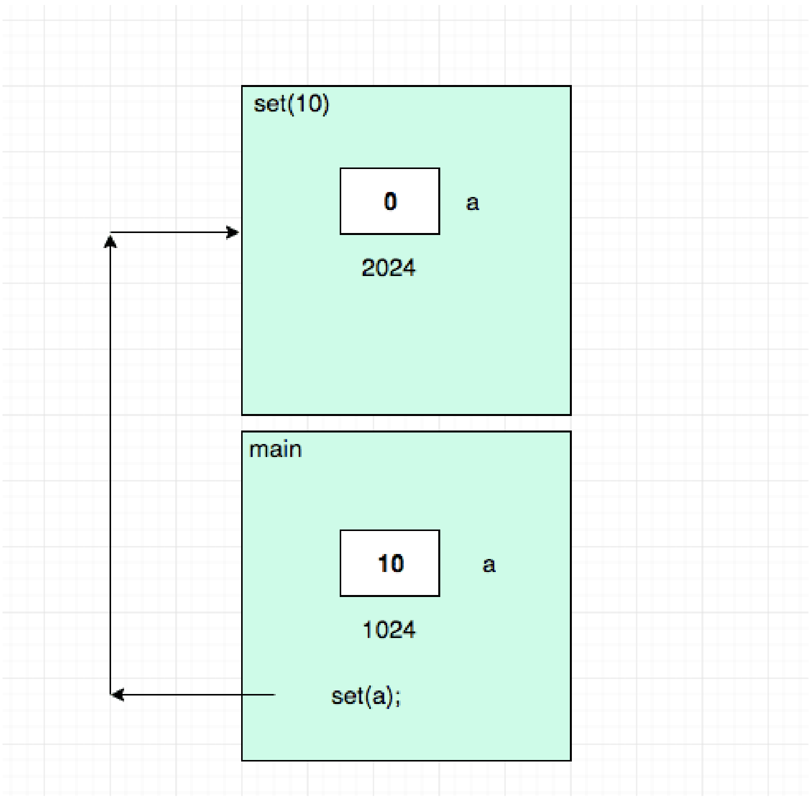


```c
//Call by value example
#include<stdio.h>
//set the argument value to 0
void set(int a)
{
    a = 0;
    printf("Set : In set function a = %d\n",a);
}
int main()
{
    int a = 10;
    printf("Main : Before calling set function a = %d\n",a);
    set(a);
    printf("Main : After calling set function a = %d\n",a);
    return 0;
}
```

OUTPUT:



```
student@student-HP-245-G6-Notebook-PC:~$ gcc callbyvalue1.c
student@student-HP-245-G6-Notebook-PC:~$ ./a.out
Main : Before calling set function a = 10
Set : In set function a = 0
Main : After calling set function a = 10
```

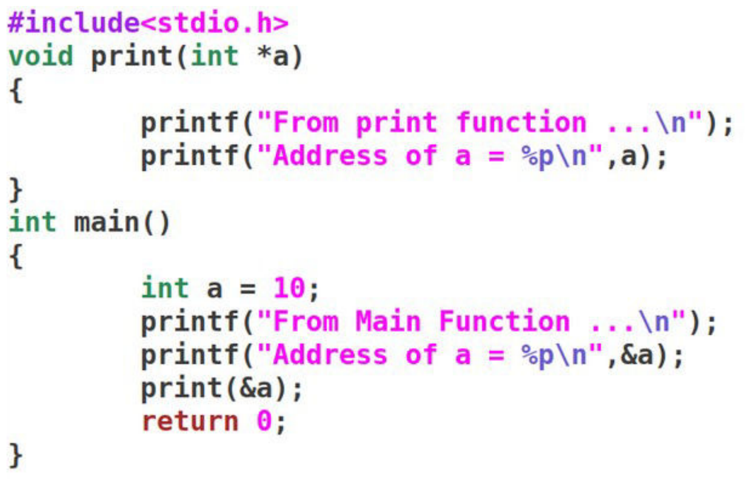


CALL BY REFERENCE

In call by reference, we pass the address of a variable.

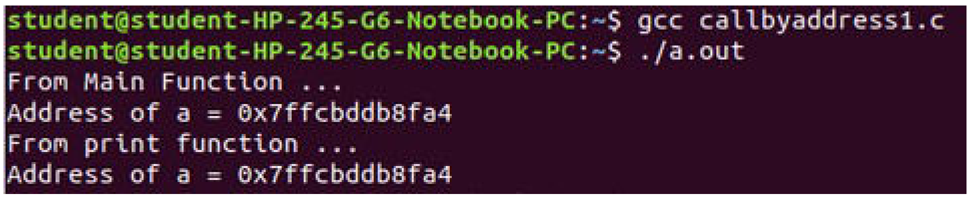
So, if we modify the value, it will affect the original value of a variable.

Program:



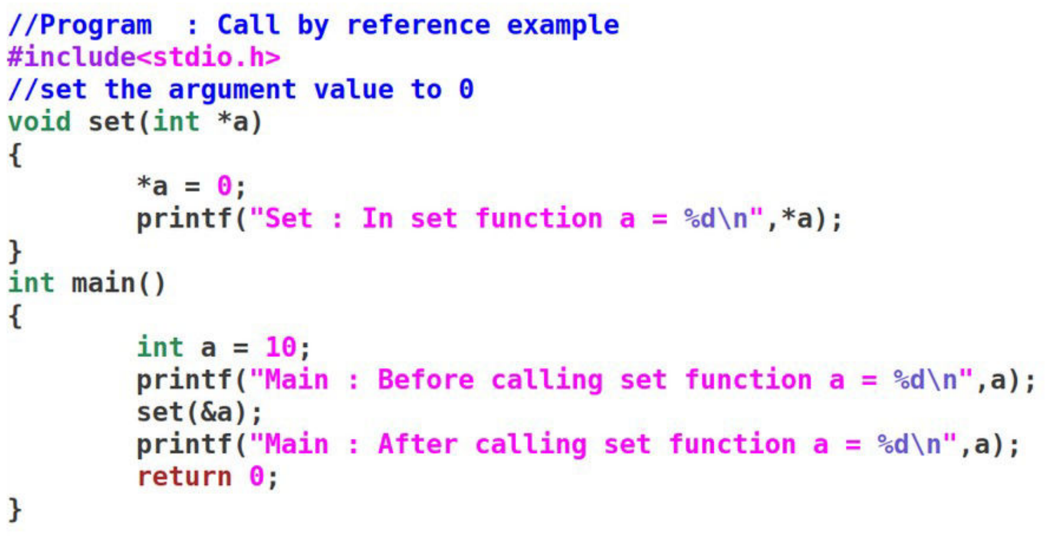
```c
#include<stdio.h>
void print(int *a)
{
        printf("From print function ...\n");
        printf("Address of a = %p\n",a);
}
int main()
{
        int a = 10;
        printf("From Main Function ...\n");
        printf("Address of a = %p\n",&a);
        print(&a);
        return 0;
}
```

Output:



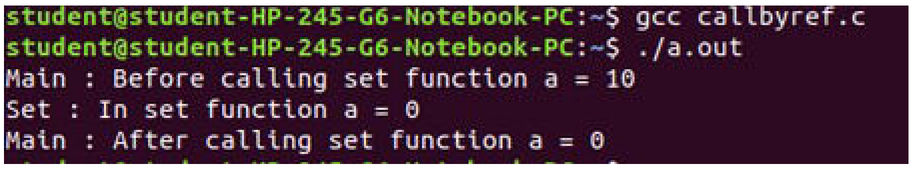
```
student@student-HP-245-G6-Notebook-PC:~$ gcc callbyaddress1.c
student@student-HP-245-G6-Notebook-PC:~$ ./a.out
From Main Function ...
Address of a = 0x7ffcbddb8fa4
From print function ...
Address of a = 0x7ffcbddb8fa4
```

We can observe that the address of **a** in the main function and the address of **a** in print function are same.

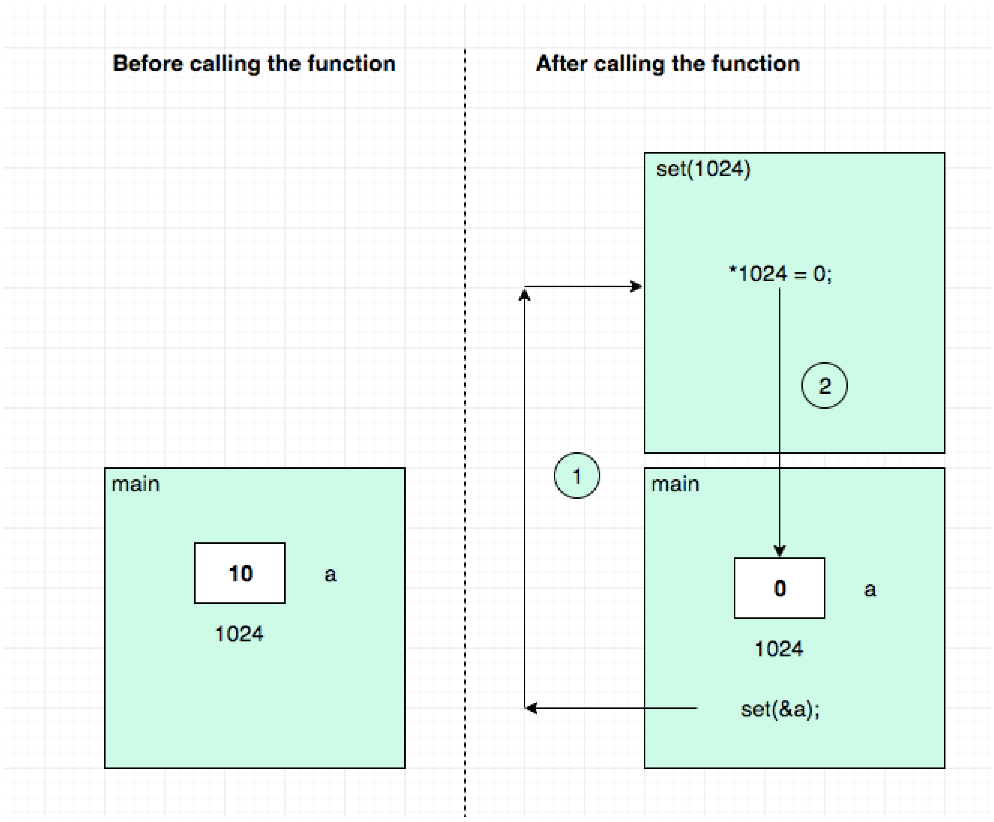


```c
//Program  : Call by reference example
#include<stdio.h>
//set the argument value to 0
void set(int *a)
{
        *a = 0;
        printf("Set : In set function a = %d\n",*a);
}
int main()
{
        int a = 10;
        printf("Main : Before calling set function a = %d\n",a);
        set(&a);
        printf("Main : After calling set function a = %d\n",a);
        return 0;
}
```

Output:



```
student@student-HP-245-G6-Notebook-PC:~$ gcc callbyref.c
student@student-HP-245-G6-Notebook-PC:~$ ./a.out
Main : Before calling set function a = 10
Set : In set function a = 0
Main : After calling set function a = 0
```



DYNAMIC MEMORY ALLOCATION

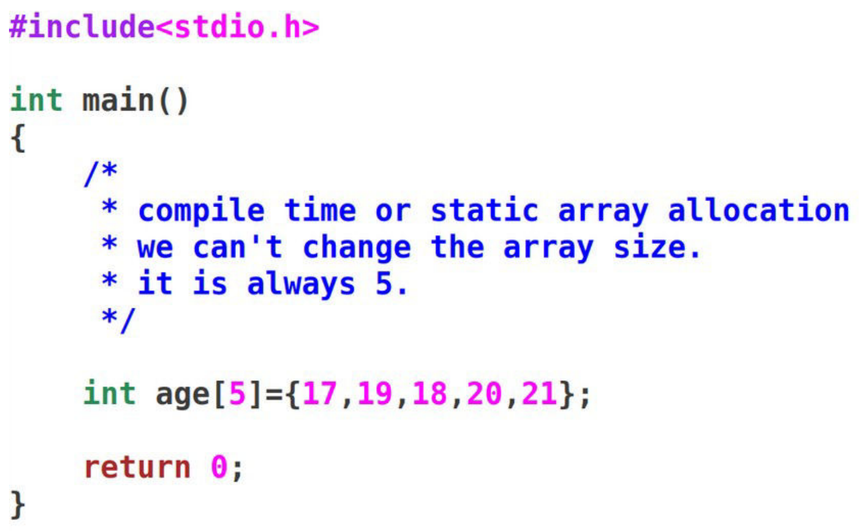
Using pointers, we can allocate memory dynamically at runtime.

Let's see the difference between static memory allocation and dynamic memory allocation.

STATIC MEMORY ALLOCATION:

If we decide the final size of a variable or an array before running the program, it will be called as static memory allocation. It is also called as compile-time memory allocation.

We can't change the size of a variable which is allocated at compile-time.



```c
#include<stdio.h>

int main()
{
    /*
     * compile time or static array allocation
     * we can't change the array size.
     * it is always 5.
     */

    int age[5]={17,19,18,20,21};

    return 0;
}
```

In the above program, we can't change the age array size based on our requirement. It always stores age of 5 students throughout the program execution. So if student size increases more than 5, we have to change the age array size manually every time.

To avoid that, we can allocate memory dynamically at runtime based on our requirement.

DYNAMIC MEMORY ALLOCATION:

At runtime, we can create, resize, deallocate the memory based on our requirement using pointers.

In the above code example, if student size increases, we can re-size the memory at runtime.If we don't need the age data at some point in time, we can free the memory area occupied by the age array.

C language provides different functions to achieve runtime memory allocation,

To allocate memory------malloc calloc

To resize the memory------realloc

To deallocate memory------free

malloc function

Using malloc function, we can create memory dynamically at runtime.

malloc is declared in <stdlib.h> i.e. standard library header file.

To use malloc function in our program, we have to include <stdlib.h> header file.

Syntax of malloc:

```
malloc(size in bytes);
```

Example:

```
char *ptr;
ptr = malloc(10);
```

malloc will take only one argument.

where,

ptr is a pointer. It can be any type.

malloc - used to create dynamic memory

10- It will allocate 10 bytes of memory.

How does malloc work?

malloc will create the dynamic memory with given size and returns the base address to the pointer.

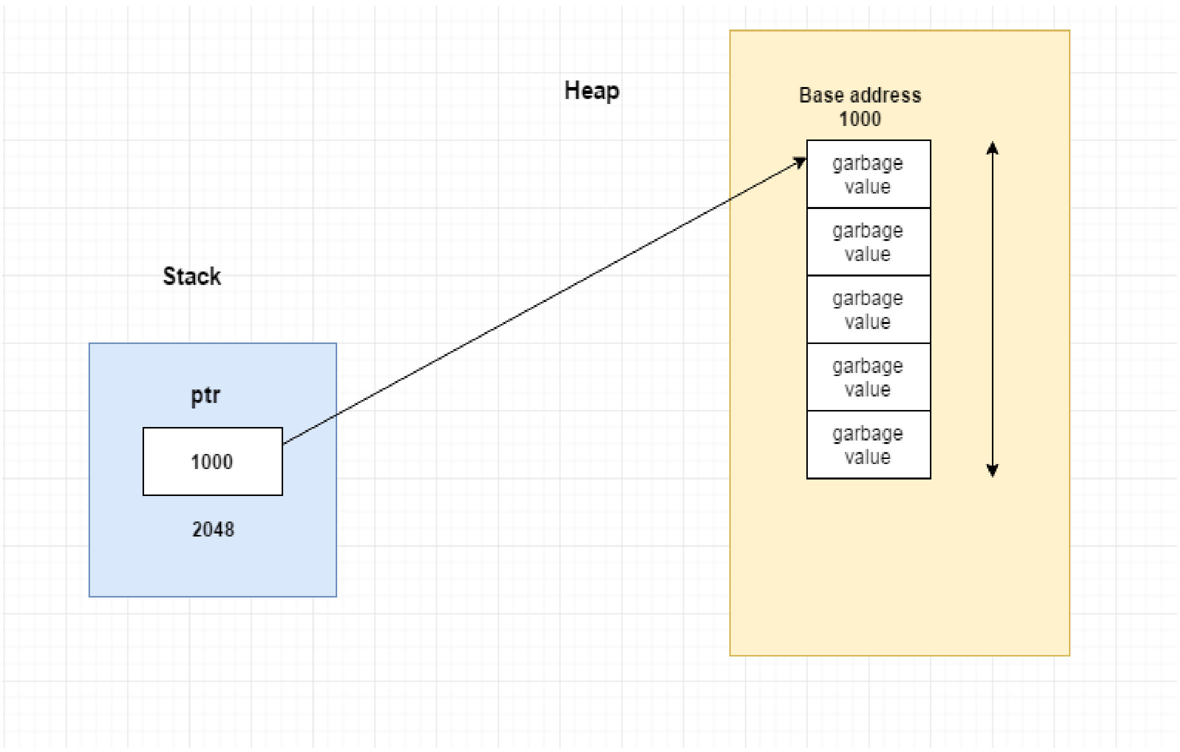
If malloc unable to create the dynamic memory, it will return NULL.

So, it is mandatory to check the pointer before using it.

Example:

```
//allocating memory for 5 integers using malloc
  int *ptr = malloc(5*sizeof(int));
```

PICTORIAL EXPLANATION:



malloc will create 20 (5 * 4) bytes of memory and return the base address to pointer variable **ptr** on success.

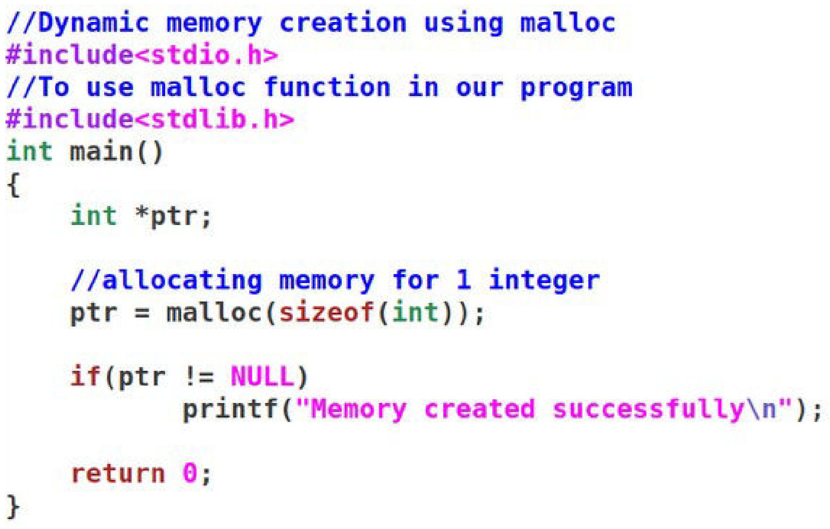
INITIALIZATION:

malloc doesn't initialize the memory area which is created dynamically.

So, the memory area will have garbage values.

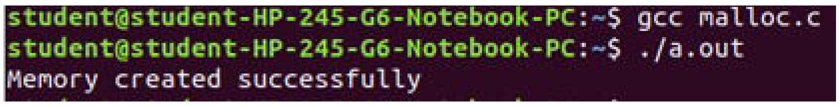
Let's create a dynamic memory using malloc

Example:



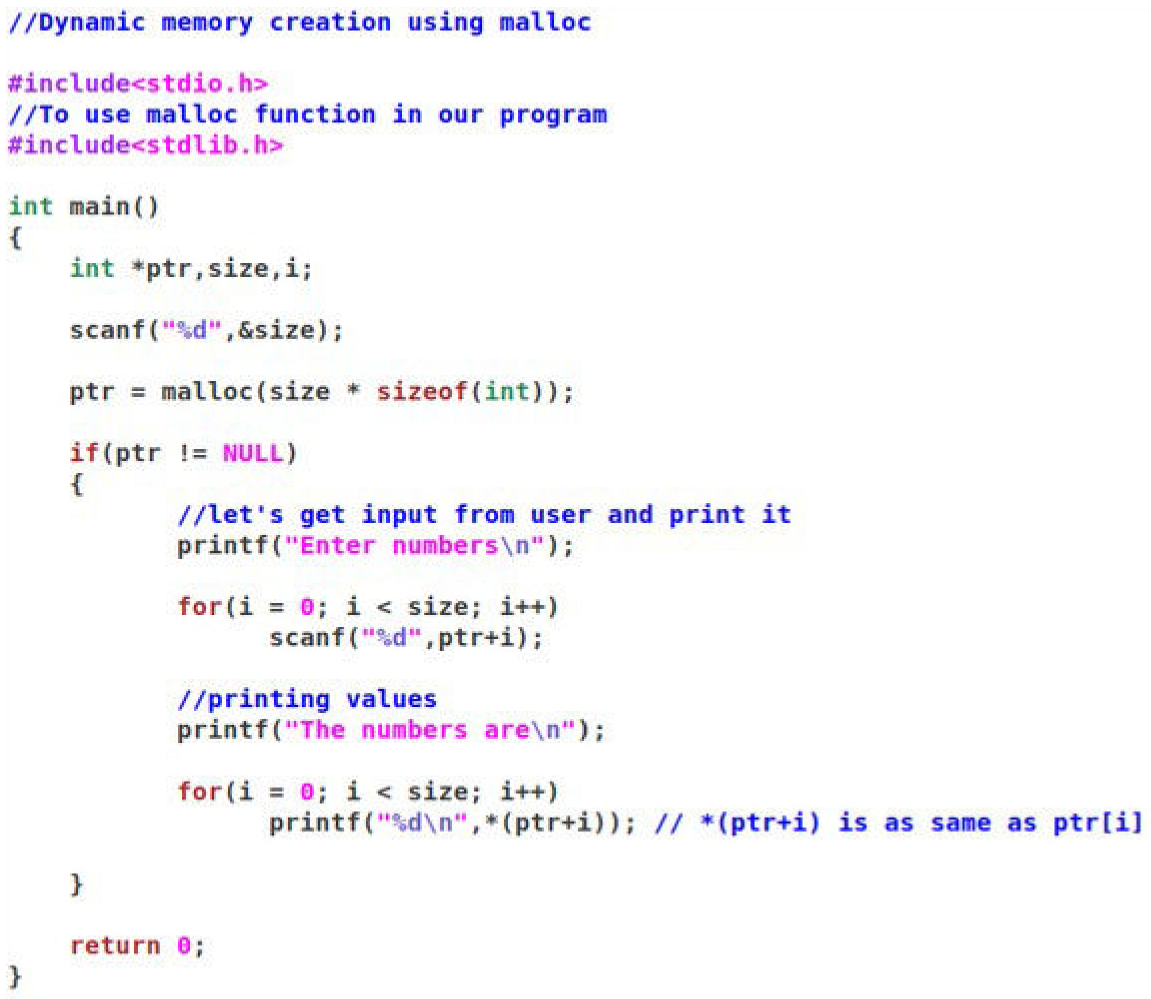
```
//Dynamic memory creation using malloc
#include<stdio.h>
//To use malloc function in our program
#include<stdlib.h>
int main()
{
    int *ptr;

    //allocating memory for 1 integer
    ptr = malloc(sizeof(int));

    if(ptr != NULL)
           printf("Memory created successfully\n");

    return 0;
}
```

output:



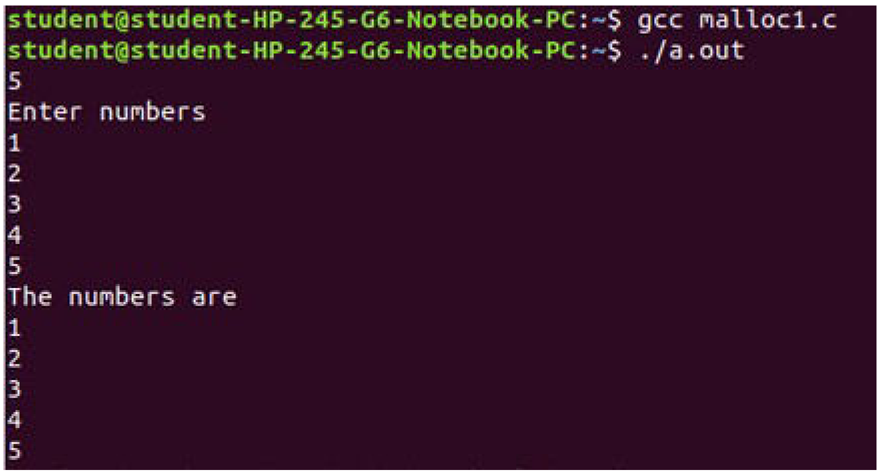
```
student@student-HP-245-G6-Notebook-PC:~$ gcc malloc.c
student@student-HP-245-G6-Notebook-PC:~$ ./a.out
Memory created successfully
```

Let's get size input from the user and allocate the dynamic memory using malloc.



```c
//Dynamic memory creation using malloc

#include<stdio.h>
//To use malloc function in our program
#include<stdlib.h>

int main()
{
    int *ptr,size,i;

    scanf("%d",&size);

    ptr = malloc(size * sizeof(int));

    if(ptr != NULL)
    {
           //let's get input from user and print it
           printf("Enter numbers\n");

           for(i = 0; i < size; i++)
                 scanf("%d",ptr+i);

           //printing values
           printf("The numbers are\n");

           for(i = 0; i < size; i++)
                 printf("%d\n",*(ptr+i)); // *(ptr+i) is as same as ptr[i]

    }

    return 0;
}
```

OUTPUT:



```
student@student-HP-245-G6-Notebook-PC:~$ gcc malloc1.c
student@student-HP-245-G6-Notebook-PC:~$ ./a.out
5
Enter numbers
1
2
3
4
5
The numbers are
1
2
3
4
5
```

calloc function

Using calloc function, we can create memory dynamically at runtime.

calloc is declared in <stdlib.h> i.e. standard library header file.

To use calloc function in our program, we have to include <stdlib.h> header file.

Syntax of calloc:

```
calloc(number of elements, size of the element);
```

Example:

```
int *ptr;
ptr = calloc(n,size);
```

calloc will take two arguments.

where,

ptr is a pointer. It can be any type.

calloc - used to create dynamic memory

n- number of elements

size- size of the elements

How does calloc work?

It will return the base address of an allocated memory to the pointer variable, on success.

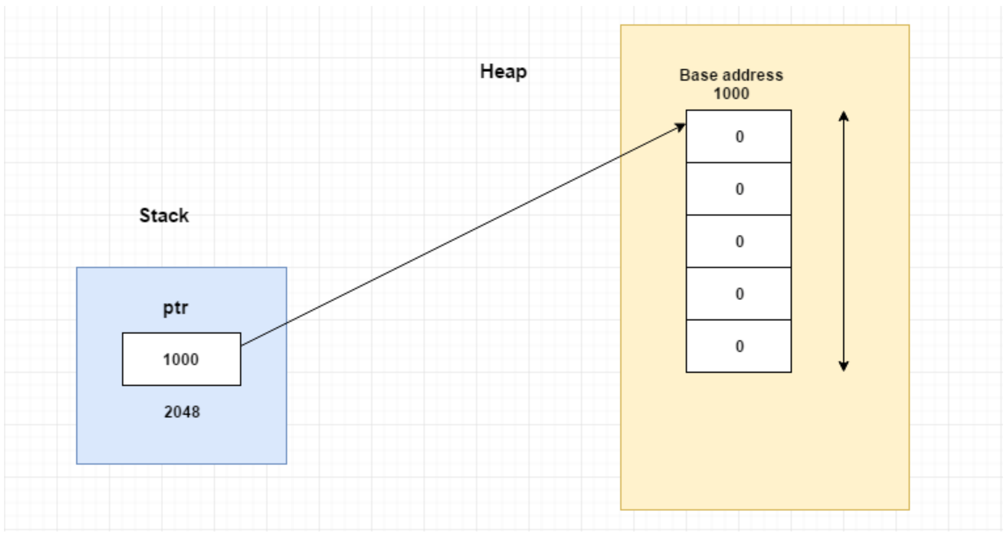
If calloc unable to create the dynamic memory, it will return NULL.

So, it is mandatory to check the pointer before using it.

Example:

```
//allocating memory for 5 integers using calloc
int *ptr = calloc(5,sizeof(int));
```

Pictorial Explanation:



calloc will create 20 (5 * 4) bytes of memory and return the base address to pointer variable ptr on success.

Initialization:

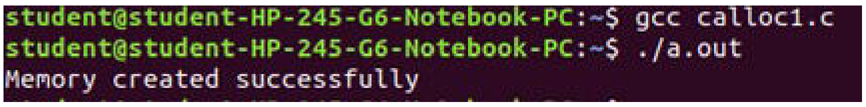
calloc will initialize the memory to zero.

So, the allocated memory area will be set to 0.

Let's create a dynamic memory using calloc:

```c
//Dynamic memory creation using calloc
 #include<stdio.h>
//To use calloc function in our program
#include<stdlib.h>
int main()
{
    int *ptr;

    //allocating memory for 1 integer
    ptr = calloc(1,sizeof(int));

    if(ptr != NULL)
           printf("Memory created successfully\n");

    return 0;
}
```

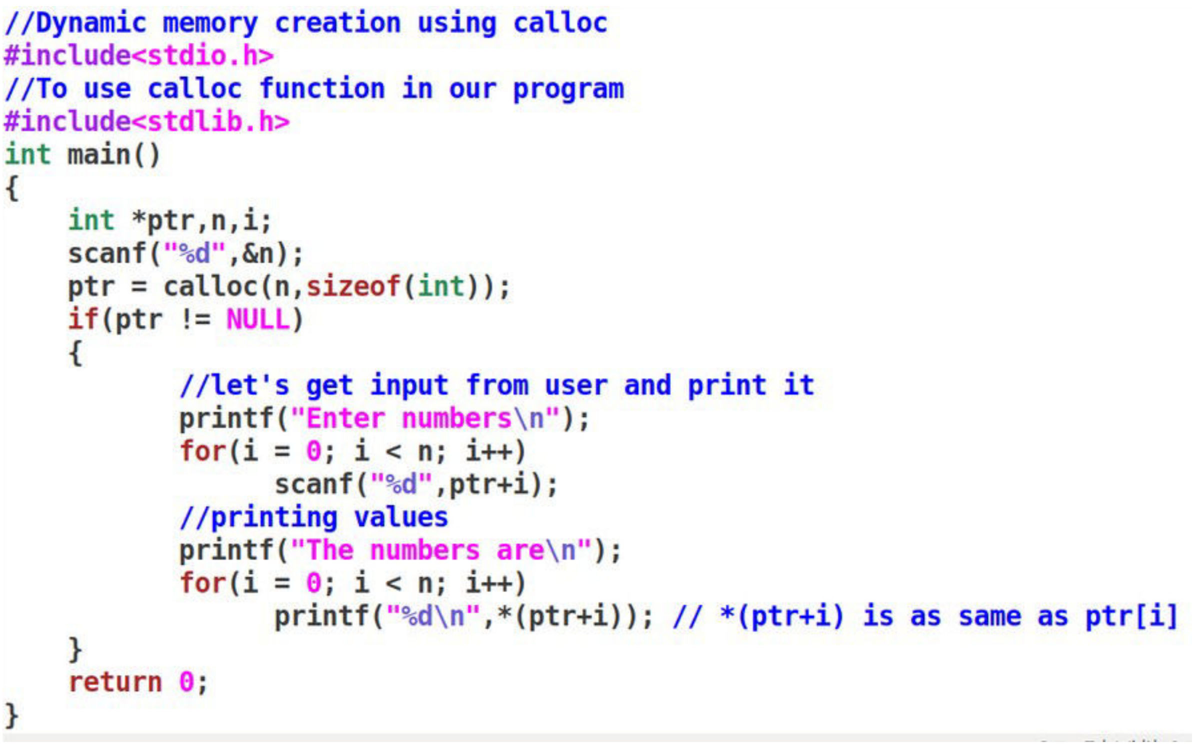
Output:



```
student@student-HP-245-G6-Notebook-PC:~$ gcc calloc1.c
student@student-HP-245-G6-Notebook-PC:~$ ./a.out
Memory created successfully
```

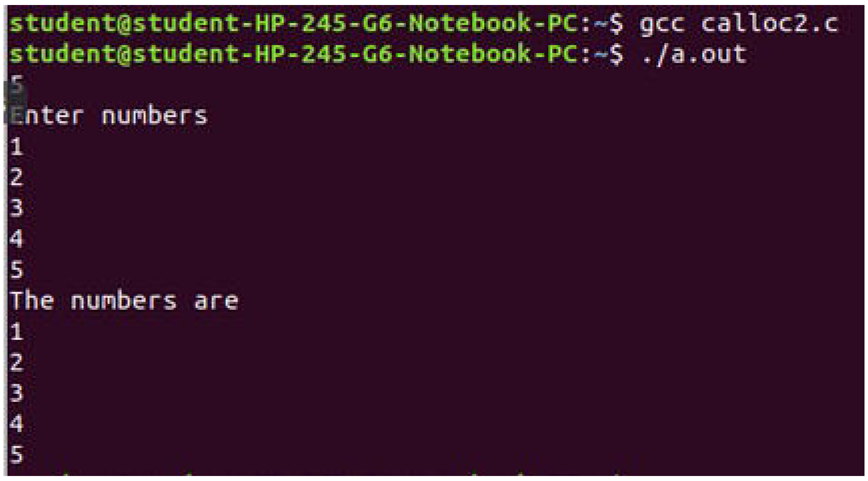
Let's get size input from the user and allocate the dynamic memory using calloc.

Example:



```
//Dynamic memory creation using calloc
#include<stdio.h>
//To use calloc function in our program
#include<stdlib.h>
int main()
{
    int *ptr,n,i;
    scanf("%d",&n);
    ptr = calloc(n,sizeof(int));
    if(ptr != NULL)
    {
            //let's get input from user and print it
            printf("Enter numbers\n");
            for(i = 0; i < n; i++)
                  scanf("%d",ptr+i);
            //printing values
            printf("The numbers are\n");
            for(i = 0; i < n; i++)
                  printf("%d\n",*(ptr+i)); // *(ptr+i) is as same as ptr[i]
    }
    return 0;
}
```

Output:



```
student@student-HP-245-G6-Notebook-PC:~$ gcc calloc2.c
student@student-HP-245-G6-Notebook-PC:~$ ./a.out
5
Enter numbers
1
2
3
4
5
The numbers are
1
2
3
4
5
```

malloc vs calloc

Arguments

malloc:

malloc takes only one argument.
malloc(size in bytes);

calloc:

calloc takes two arguments.
calloc(number of elements, size of the element);

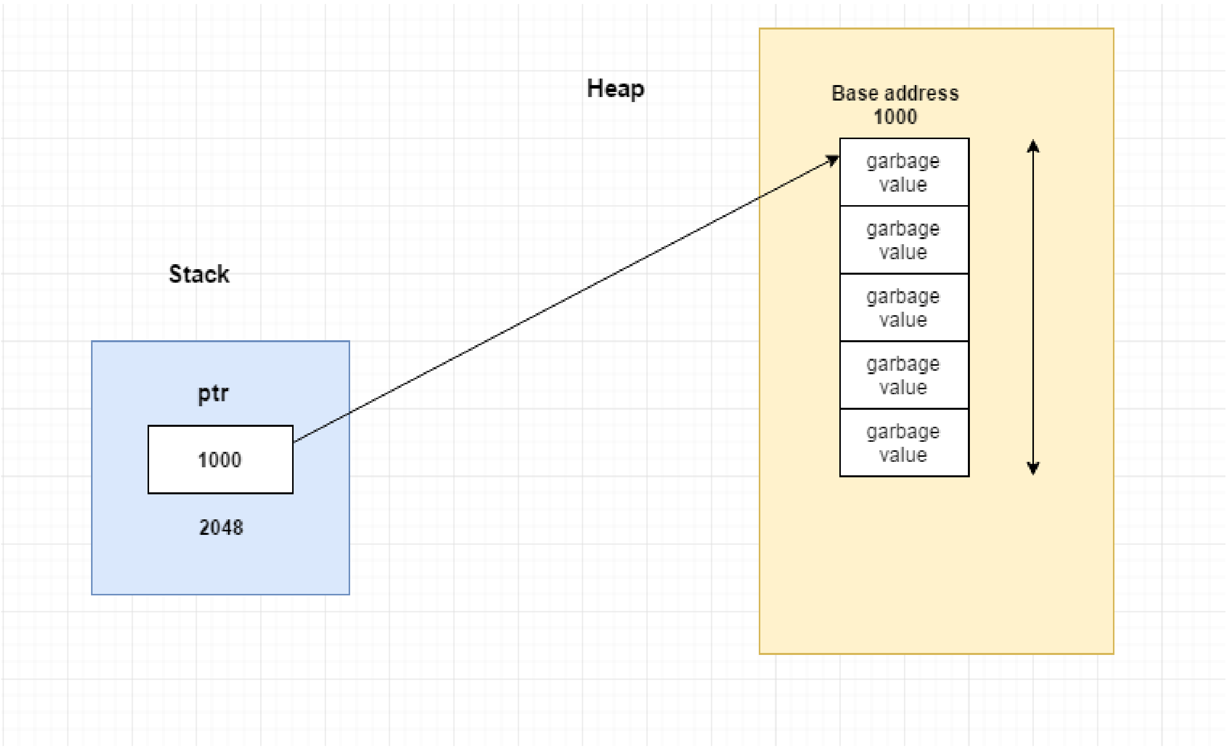
Initialization:

malloc:

malloc doesn't initialize the memory area.

The allocated memory area will have garbage values.

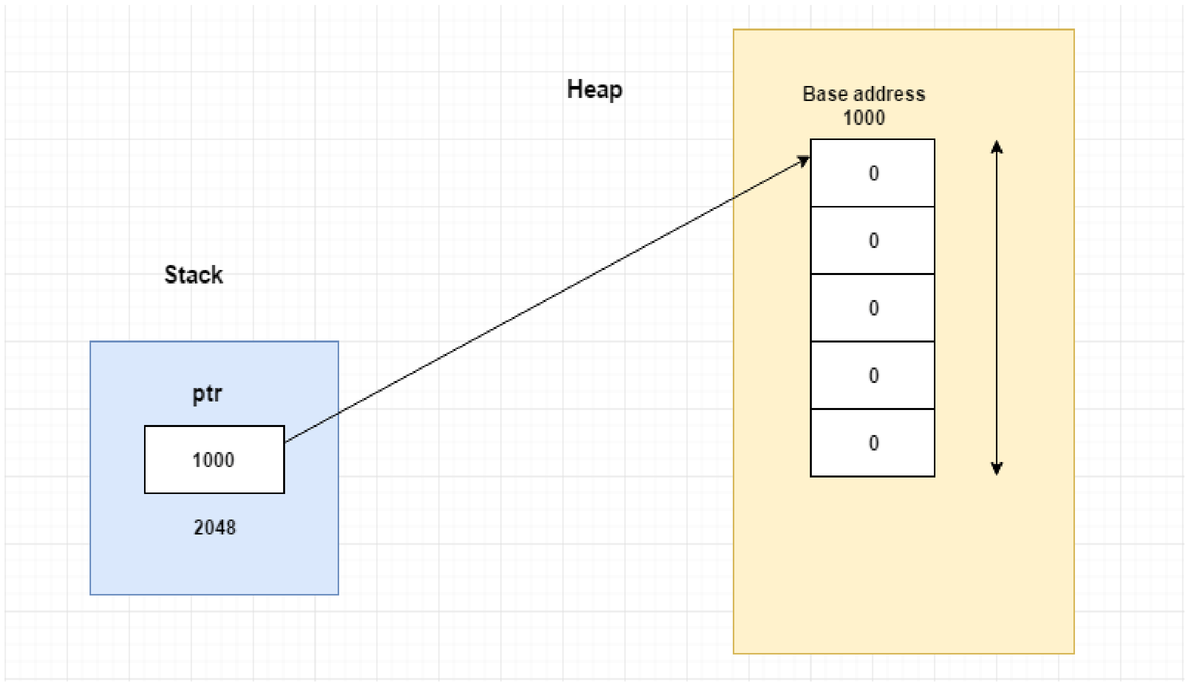
Pictorial Explanation:



calloc:

calloc initialize the memory area to zero.

Pictorial Explanation:



realloc in c

Use of realloc function:

Using realloc function, we can resize the memory area which is already created by malloc or calloc.

It's is also declared in stdlib.h library.

Limitation

If the memory area is not created dynamically using malloc or calloc, then the behavior of the realloc function is undefined.

realloc syntax

```
realloc(ptr, new size);
```

Where,

realloc- used to resize the memory area which is pointed by ptr.

ptr- the name of the pointer variable which needs to be resized.

new size- the new size of the memory area. It can be smaller or bigger than the actual size.

Return value of realloc

If realloc request success, it will return a pointer to the newly allocated memory.

Otherwise, it will return NULL.

Example

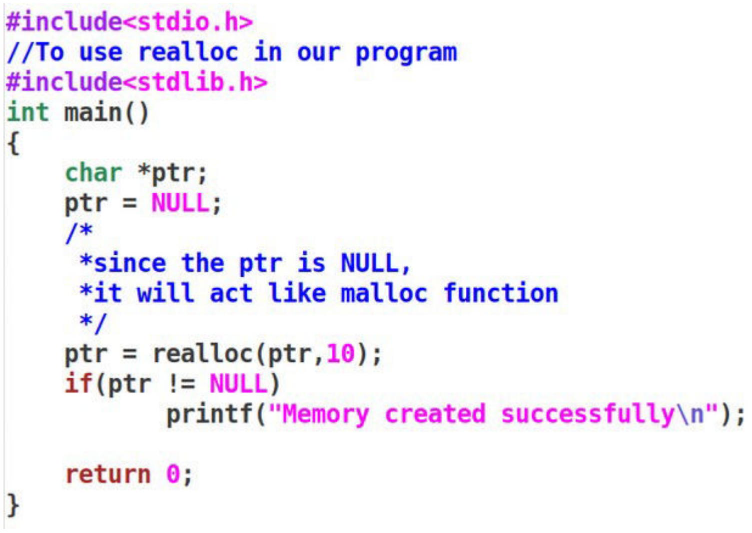
Changing size from 100 bytes to 1000 bytes using realloc.

```c
char *ptr;
ptr = malloc(100);
ptr = realloc(ptr,1000);
```

How realloc works?

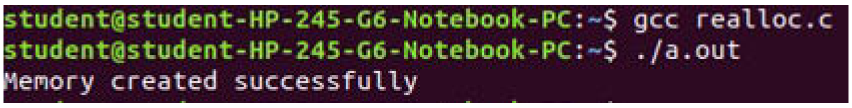
realloc will act as malloc if the pointer variable is NULL.

Example:



```
#include<stdio.h>
//To use realloc in our program
#include<stdlib.h>
int main()
{
    char *ptr;
    ptr = NULL;
    /*
     *since the ptr is NULL,
     *it will act like malloc function
     */
    ptr = realloc(ptr,10);
    if(ptr != NULL)
           printf("Memory created successfully\n");

    return 0;
}
```

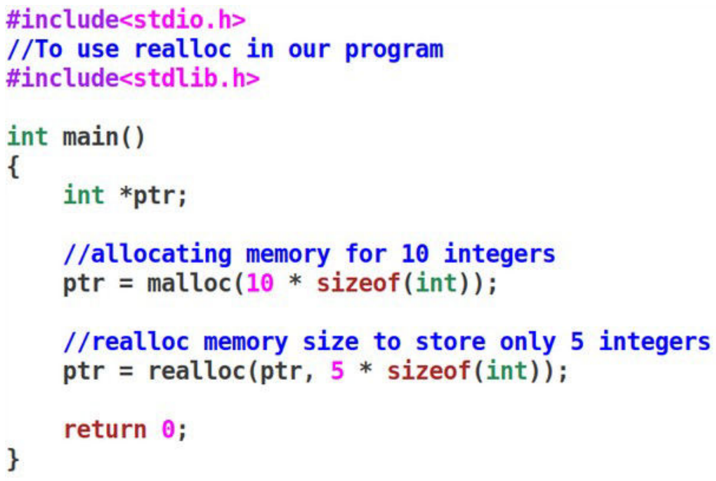
output:



```
student@student-HP-245-G6-Notebook-PC:~$ gcc realloc.c
student@student-HP-245-G6-Notebook-PC:~$ ./a.out
Memory created successfully
```

realloc smaller size

2.If the given size is smaller than the actual, it will simply reduce the memory size.

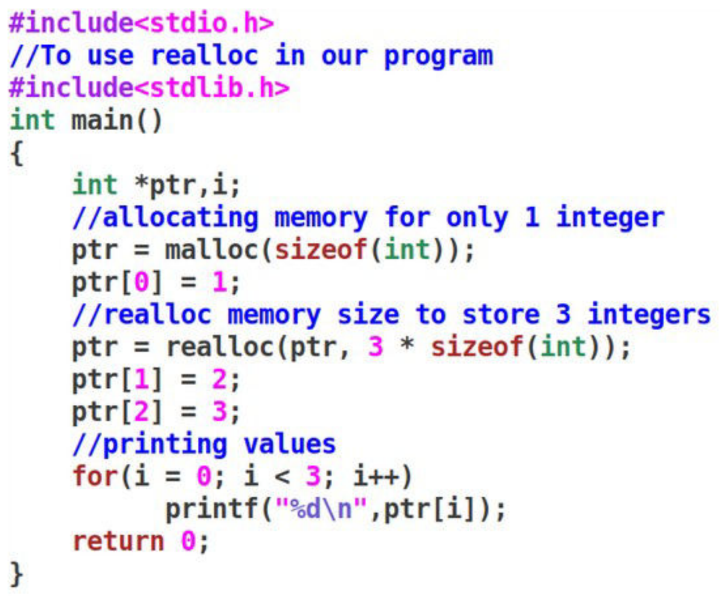


```
#include<stdio.h>
//To use realloc in our program
#include<stdlib.h>

int main()
{
    int *ptr;

    //allocating memory for 10 integers
    ptr = malloc(10 * sizeof(int));

    //realloc memory size to store only 5 integers
    ptr = realloc(ptr, 5 * sizeof(int));

    return 0;
}
```

3.If the given size is bigger than the actual, it will check whether it can expand the already available memory.

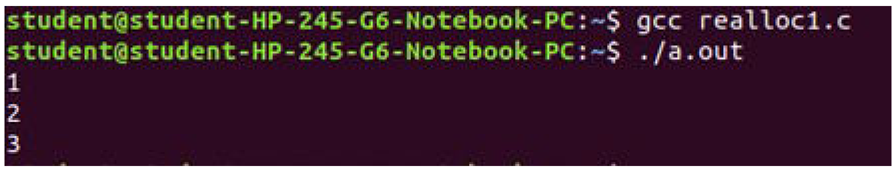
If it is possible it will simply resize the memory.

Example:



```c
#include<stdio.h>
//To use realloc in our program
#include<stdlib.h>
int main()
{
    int *ptr,i;
    //allocating memory for only 1 integer
    ptr = malloc(sizeof(int));
    ptr[0] = 1;
    //realloc memory size to store 3 integers
    ptr = realloc(ptr, 3 * sizeof(int));
    ptr[1] = 2;
    ptr[2] = 3;
    //printing values
    for(i = 0; i < 3; i++)
          printf("%d\n",ptr[i]);
    return 0;
}
```

output:



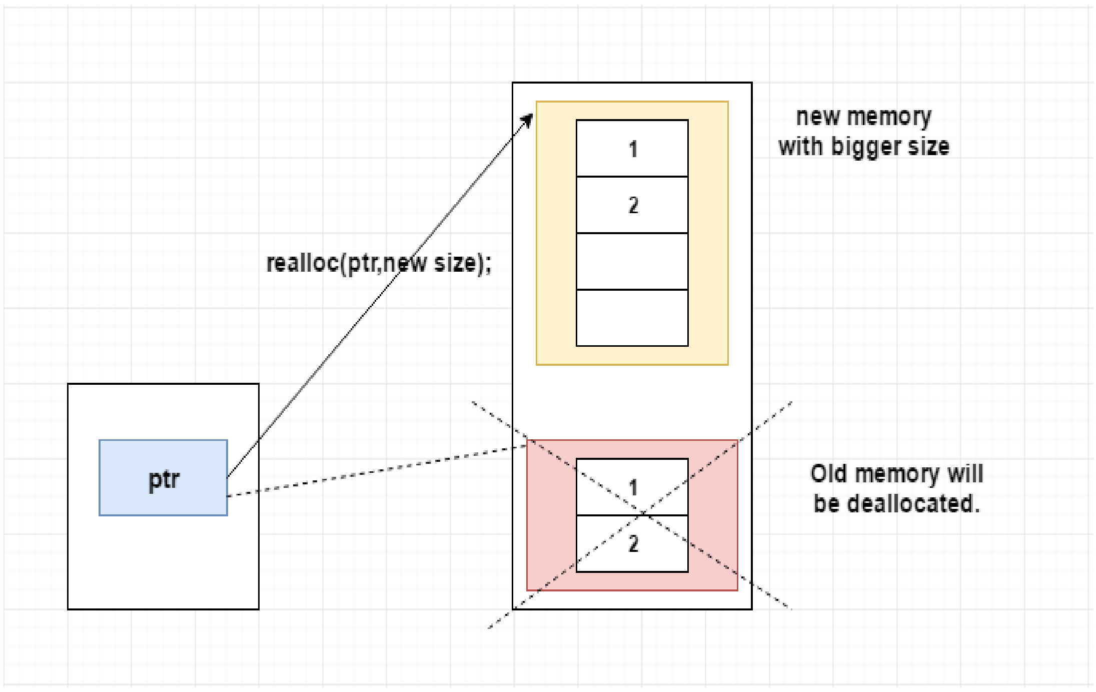
```
student@student-HP-245-G6-Notebook-PC:~$ gcc realloc1.c
student@student-HP-245-G6-Notebook-PC:~$ ./a.out
1
2
3
```

Otherwise, realloc will create the new block of memory with the larger size and copy the old data to that newly allocated memory and than it will deallocate the old memory area.

For example assume, we have allocated memory for 2 integers.We are going to increase the memory size to store 4 integers.

If realloc unable to expand the memory size, it will do something like below.

Pictorial representation:

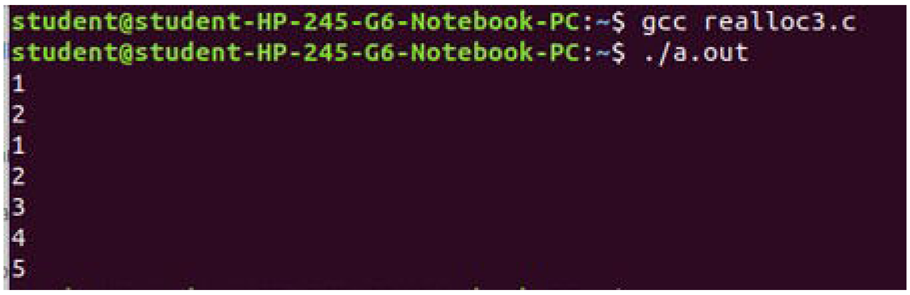


Sample Program

Example:

```
#include<stdio.h>
//To use realloc in our program
#include<stdlib.h>
int main()
{
    int *ptr,size,i;
    /*
     * Let's create memory for 2 integers
     * size = 2
     */
    size = 2;
    ptr = malloc(size * sizeof(int));
    // *(ptr + i) === ptr[i]
    *(ptr + 0) = 1;
    *(ptr + 1) = 2;
    //printing elements
    for(i = 0; i < size; i++)
          printf("%d\n",ptr[i]);
    /*
     * Let's change the size to store 3 more integers
     * size = 5
     */
    size = 5;
    ptr = realloc(ptr, size * sizeof(int));
    *(ptr + 2) = 3;
    *(ptr + 3) = 4;
    *(ptr + 4) = 5;
    for(i = 0; i < size; i++)
          printf("%d\n",ptr[i]);
    return 0;
}
```

output:



```
student@student-HP-245-G6-Notebook-PC:~$ gcc realloc3.c
student@student-HP-245-G6-Notebook-PC:~$ ./a.out
1
2
1
2
3
4
5
```

free

It's mandatory to deallocate the memory area which has created dynamically when the memory is no longer needed.

Why do we need to deallocate the dynamic memory?

When we try to create the dynamic memory, the memory will be created in the heap section.

Memory Leak:

If we don't deallocate the dynamic memory, it will reside in the heap section.It is also called memory leak.

It will reduce the system performance by reducing the amount of available memory.

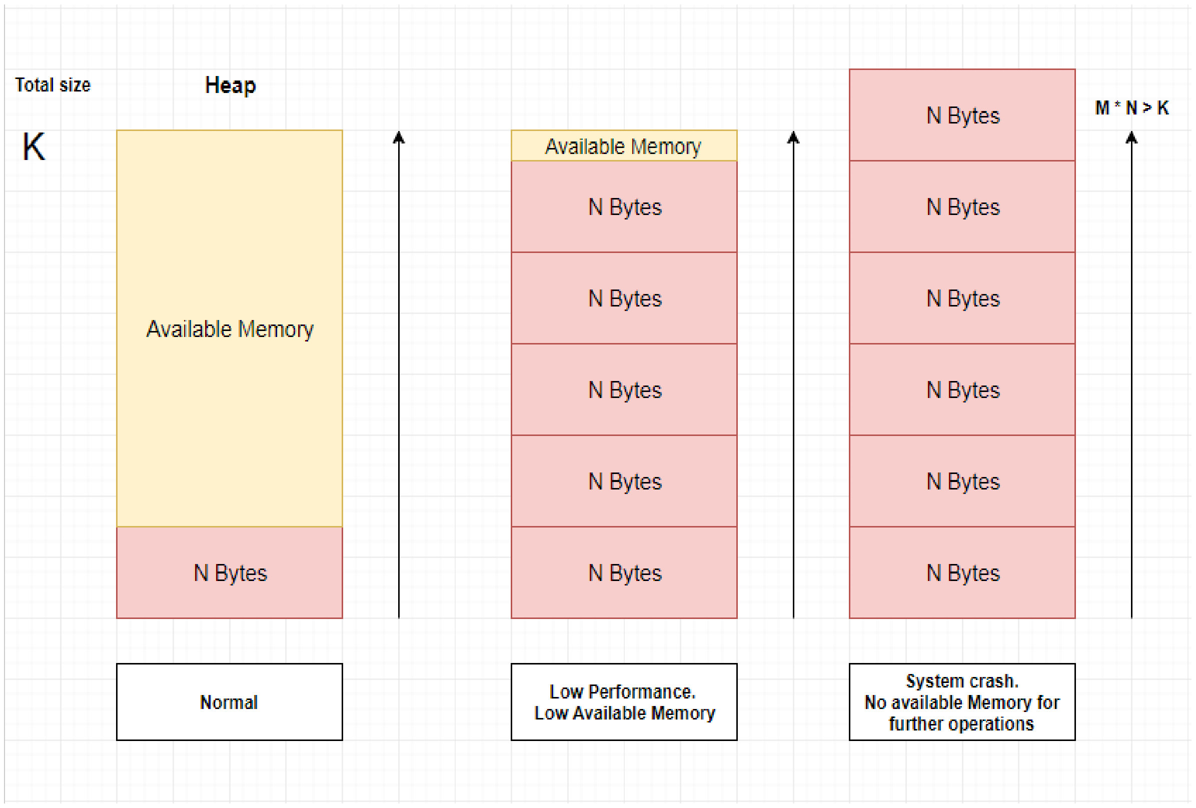
Let's assume total heap size as K.

If we allocate N byte of memory dynamically, it will consume N bytes of memory in heap section.

When the particular piece of code executed M number of time, then M * N bytes of memory will be consumed by our program.

At some point in time (M * N > K), the whole heap memory will be consumed by the program it will result in the system crash due to low available memory.

Pictorial Explanation



So, it is programmers responsibility to deallocate the dynamic memory which is no longer needed.

How to deallocate the dynamic memory?

Using free() function, we can deallocate the dynamic memory.

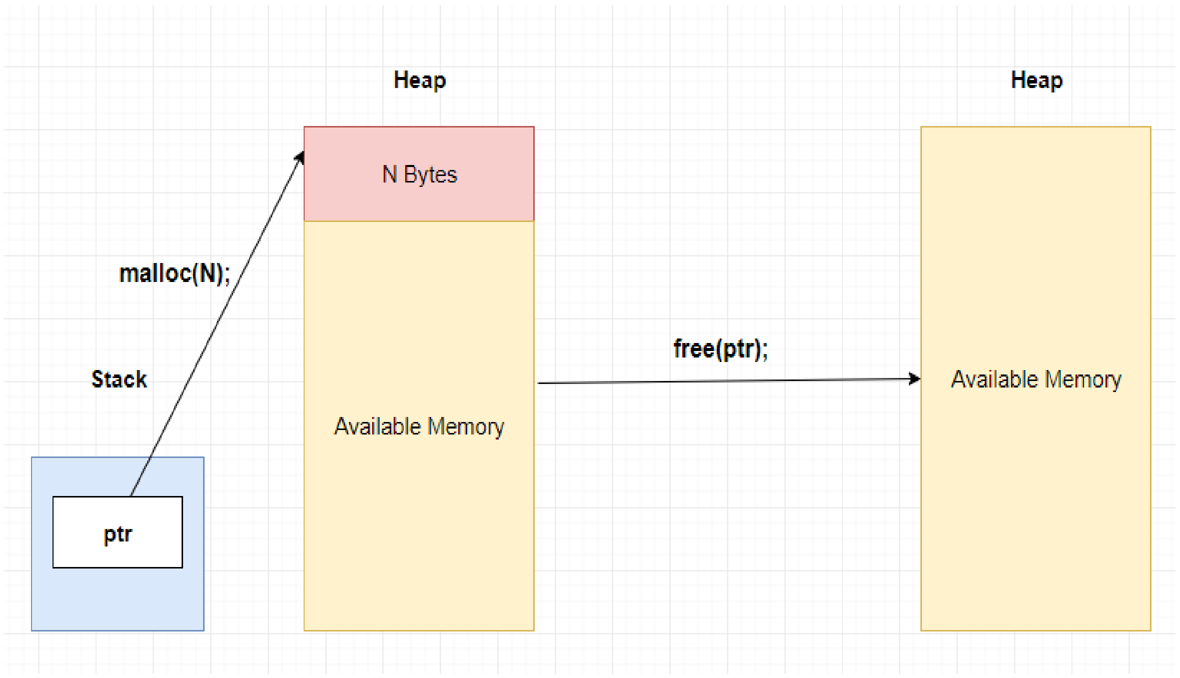
Syntax of free:

```
free(ptr);
```

Example:

```
char *ptr;

  ptr = malloc(N);
  //do something
  free(ptr);
```

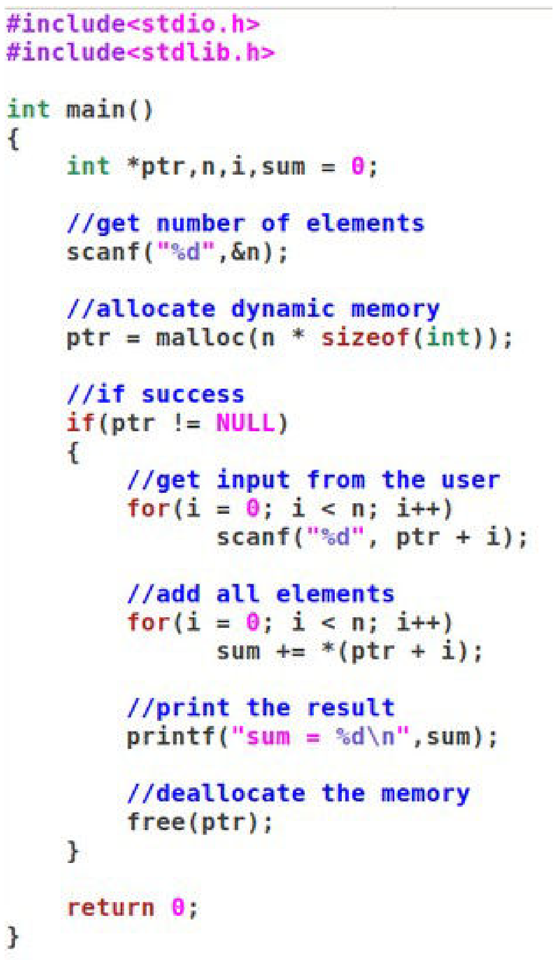
Pictorial Explanation:



Sample program

Sum of N numbers. Get N from the user and use dynamic memory to store the inputs.

Example:



```c
#include<stdio.h>
#include<stdlib.h>

int main()
{
    int *ptr,n,i,sum = 0;

    //get number of elements
    scanf("%d",&n);

    //allocate dynamic memory
    ptr = malloc(n * sizeof(int));

    //if success
    if(ptr != NULL)
    {
        //get input from the user
        for(i = 0; i < n; i++)
              scanf("%d", ptr + i);

        //add all elements
        for(i = 0; i < n; i++)
              sum += *(ptr + i);

        //print the result
        printf("sum = %d\n",sum);

        //deallocate the memory
        free(ptr);
    }

    return 0;
}
```

Double free is undefined

If we free the same pointer two or more time, then the behavior is undefined.

So, if we free the same pointer which is freed already, the program will stop its execution.

Example:

```
char *ptr;
ptr = malloc(10);
free(ptr);
```

Command Line Arguments in C

- In C programming language, command line arguments are the data values that are passed from command line to our program.
- Using command line arguments we can control the program execution from the outside of the program.
- Generally, all the command line arguments are handled by the main() method.
- Generally, the command line arguments can be understood as follows...

Command line arguments are the parameters passing to main() method from the command line.

- When command line arguments are passed main() method receives them with the help of two formal parameters and they are,

- **int argc**
- **char *argv[ ]**

int argc **-** It is an integer argument used to store the count of command line arguments are passed from the command line.

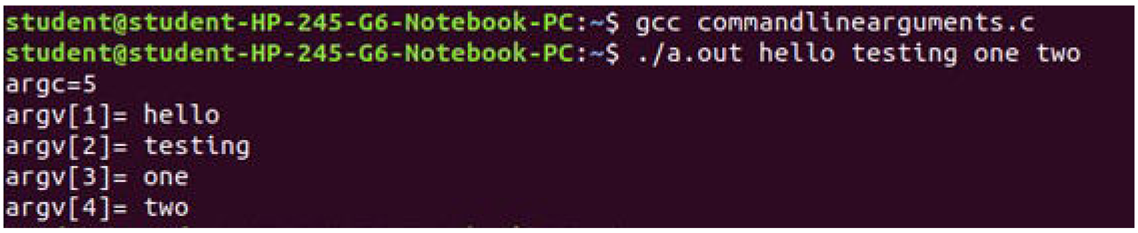
char *argv[ ] **-** It is a character pointer array used to store the actual values of command line arguments are passed from the command line.

- The command line arguments are separated with **SPACE**.
- Always the first command line argument is file path.
- Only string values can be passed as command line arguments.
- All the command line arguments are stored in a character pointer array called **argv[ ]**.
- Total count of command line arguments including file path argument is stored in a integer parameter called **argc**.

Example Program to access <u>command line arguments</u> in C.

```
#include<stdio.h>
int main(int argc, char *argv[])
{
        printf("argc=%d\n",argc);
        for(int i=1; i<argc ; i++)
        {
            printf("argv[%d]= %s \n", i, argv[i]);
        }
        return 0;
}
```

Output:



```
student@student-HP-245-G6-Notebook-PC:~$ gcc commandlinearguments.c
student@student-HP-245-G6-Notebook-PC:~$ ./a.out hello testing one two
argc=5
argv[1]= hello
argv[2]= testing
argv[3]= one
argv[4]= two
```

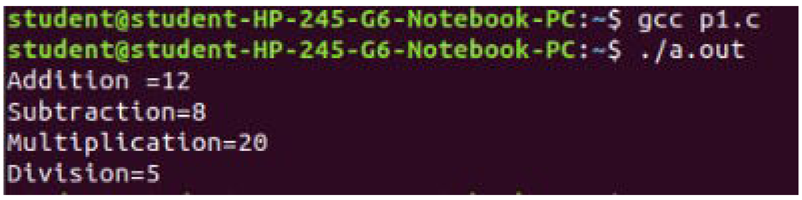
PROGRAMMING EXAMPLES

1. Write a C program to illustrate the use of pointers in arithmetic operations?

Program:

```c
//PROGRAM TO ILLUSTRATE THE USE OF POINTERS IN ARITHMETIC OPERATIONS
#include<stdio.h>
void main()
{
		int a=10,b=2;
		int *ptr1,*ptr2;
		ptr1=&a;
		ptr2=&b;
		printf("Addition =%d\n",(*ptr1)+(*ptr2));
		printf("Subtraction=%d\n",(*ptr1)-(*ptr2));
		printf("Multiplication=%d\n",(*ptr1)*(*ptr2));
		printf("Division=%d\n",(*ptr1)/(*ptr2));
}
```

Output:



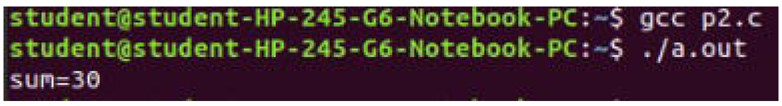
```
student@student-HP-245-G6-Notebook-PC:~$ gcc p1.c
student@student-HP-245-G6-Notebook-PC:~$ ./a.out
Addition =12
Subtraction=8
Multiplication=20
Division=5
```

2. Write a C program using pointers to compute the sum of all elements stored in an array.

Program:

```c
//program using pointers to compute the sum of all elements stored in an array
#include<stdio.h>
void main()
{
	int *p,sum=0,i;
	int x[5]={5,9,6,3,7};
	i=0;
	p=x;
	while(i<5)
	{
		sum=sum +(*p);
		i++,p++;
	}
	printf("sum=%d\n",sum);
}
```

Output:



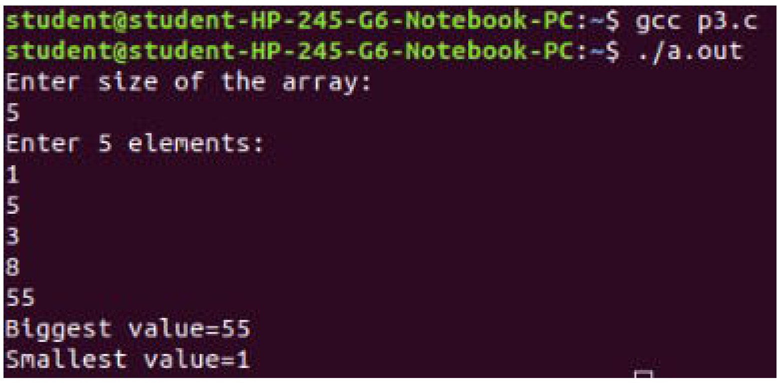
```
student@student-HP-245-G6-Notebook-PC:~$ gcc p2.c
student@student-HP-245-G6-Notebook-PC:~$ ./a.out
sum=30
```

3. Write a C program to find smallest and biggest numbers of the given number using pointers?

Program:

```c
//program to find smallest and biggest numbers of the given number using pointers
#include<stdio.h>
void main()
{
        int a[50],*ptr,n,i,small,big;
        printf("Enter size of the array:\n");
        scanf("%d",&n);
        ptr=a;
        printf("Enter %d elements:\n",n);
        for(i=0;i<n;i++)
        {
                scanf("%d",(ptr+i));
        }
        big=small=*ptr;
        for(i=0;i<n;i++,ptr++)
        {
                if(big<*ptr)
                        big=*ptr;
                if(small>*ptr)
                        small=*ptr;
        }
        printf("Biggest value=%d\n",big);
        printf("Smallest value=%d\n",small);
}
```

Output:



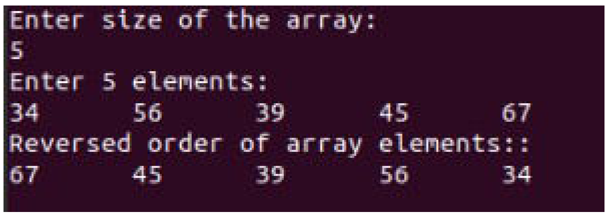
```
student@student-HP-245-G6-Notebook-PC:~$ gcc p3.c
student@student-HP-245-G6-Notebook-PC:~$ ./a.out
Enter size of the array:
5
Enter 5 elements:
1
5
3
8
55
Biggest value=55
Smallest value=1
```

4. Write a C program using pointers to read in an array of integer and point its elements in reverse order.

Program:

```
/* program using pointers to read in an array of integer
and point its elements in reverse order*/
#include<stdio.h>
void main()
{
        int a[50],*ptr,n,i;
        printf("Enter size of the array:\n");
        scanf("%d",&n);
        ptr=a;
        printf("Enter %d elements:\n",n);
        for(i=0;i<n;i++)
        {
                scanf("%d",ptr++);
        }
        printf("Reversed order of array elements::\n");
        for(i=0;i<n;i++)
        {
                printf("%d\t",*(--ptr));
        }
}
```

Output:



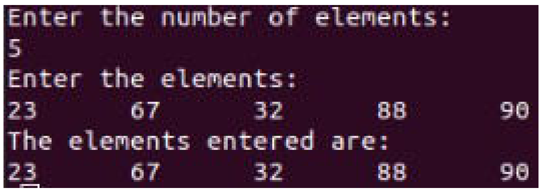
```
Enter size of the array:
5
Enter 5 elements:
34      56      39      45      67
Reversed order of array elements::
67      45      39      56      34
```

5. Write a C program to read and display array of n integers using pointers

Program:

```
//A program to read and display an array of n integers
#include<stdio.h>
void main()
{
        int i,n;
        int arr[10],*parr=arr;
        printf("Enter the number of elements:\n");
        scanf("%d",&n);
        printf("Enter the elements:\n");
        for(i=0;i<n;i++)
        {
                scanf("%d",parr+i);
        }
        printf("The elements entered are:\n");
        for(i=0;i<n;i++)
        {
                printf("%d\t",*(parr+i));
        }
}
```

Output:



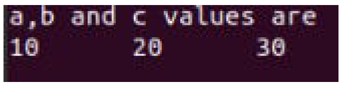
```
Enter the number of elements:
5
Enter the elements:
23      67      32      88      90
The elements entered are:
23      67      32      88      90
```

6. Write a C program using array of integer pointers?

Program:

```c
//array of integer pointers
#include<stdio.h>
void main()
{
        int *ptr[3],i;
        int a=10,b=20,c=30;
        ptr[0]=&a;
        ptr[1]=&b;
        ptr[2]=&c;
        printf("a,b and c values are\n");
        for(i=0;i<3;i++)
        {
                printf("%d\t",*ptr[i]);
        }
}
```

Output:



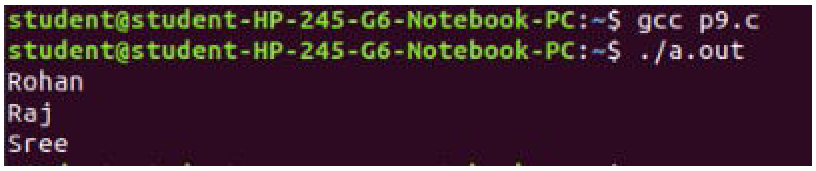
```
a,b and c values are
10      20      30
```

7. Write a C program using array of character pointers?

Program:

```c
//array of character pointers
#include<stdio.h>
void main()
{
        char *ptr[]={"Rohan","Raj","Sree"};
        int i;
        for(i=0;i<3;i++)
        {
                printf("%s\n",ptr[i]);
        }
}
```

Output:



```
student@student-HP-245-G6-Notebook-PC:~$ gcc p9.c
student@student-HP-245-G6-Notebook-PC:~$ ./a.out
Rohan
Raj
Sree
```